

目 录

初 级 篇

第 1 章 Qt 初步实践.....	2	3.5 Qt 样式表.....	74
1.1 第一个 Qt 程序.....	2	3.5.1 样式表语法.....	74
1.1.1 建立主程序.....	2	3.5.2 样式表的应用.....	76
1.1.2 建立工程.....	3	3.6 Qt 对象模型.....	79
1.1.3 编译/运行第一个 Qt 应用程序.....	8	3.6.1 元对象系统.....	79
1.1.4 第一个 Qt 程序的代码分析.....	8	3.6.2 属性系统.....	80
1.2 使用 Qt 布局管理器.....	11	3.6.3 对象树.....	83
1.3 关联操作.....	12	3.7 小结.....	86
1.4 小结.....	13	第 4 章 程序主窗口——QMainWindow.....	87
第 2 章 对话框——QDialog.....	14	4.1 QMainWindow 主窗口框架.....	87
2.1 自定义对话框.....	14	4.2 Qt 设计器绘制主窗口.....	88
2.1.1 建立新类.....	14	4.2.1 菜单.....	90
2.1.2 添加子窗口部件.....	15	4.2.2 工具栏.....	93
2.2 加入主程序.....	22	4.2.3 中心部件.....	96
2.3 Qt 内建 (built-in) 对话框.....	24	4.3 代码创建主窗口.....	98
2.4 小结.....	34	4.3.1 创建资源文件.....	98
第 3 章 基础窗口部件——QWidget.....	35	4.3.2 定义主窗口类.....	98
3.1 Qt 设计器绘制窗口部件.....	35	4.4 锚接部件.....	102
3.1.1 Qt 设计器基础.....	35	4.5 状态栏.....	105
3.1.2 绘制窗口部件.....	40	4.6 实现文本编辑器功能.....	107
3.2 程序中引入自定义窗口部件.....	47	4.7 多文档.....	118
3.2.1 直接使用方式.....	47	4.8 打印文档.....	119
3.2.2 单一继承方式.....	49	4.9 小结.....	120
3.2.3 多继承方式.....	51	第 5 章 布局管理.....	121
3.3 Qt 的信号和槽机制.....	53	5.1 Qt 布局管理器——QLayout.....	121
3.3.1 基本原理.....	53	5.1.1 Qt 布局管理器简介.....	121
3.3.2 设计信号和槽.....	55	5.1.2 布局管理器及窗口部件大小策略.....	125
3.3.3 信号和槽的自动关联.....	62	5.2 分裂器部件 QSplitter.....	132
3.4 窗口标志及几何布局.....	63	5.3 栈部件 QStackedWidget.....	134
3.4.1 窗口标志.....	64	5.4 工作空间部件 QWorkspace.....	135
3.4.2 窗口部件的几何布局.....	66	5.5 多文档区部件 QMdiArea.....	148
		5.6 小结.....	150

中 级 篇

第 6 章 2D 绘图	152
6.1 Arthur 绘图基础	152
6.1.1 绘图	152
6.1.2 绘图设备	174
6.2 坐标系统与坐标变换	175
6.2.1 坐标系统	175
6.2.2 坐标变换	175
6.3 用不同的字体	177
6.4 绘图路径——QPainterPath	180
6.5 QImage 与 QPixmap 绘图设备	182
6.5.1 QImage	182
6.5.2 QPixmap	183
6.6 组合模式绘图	192
6.7 Graphics View 框架	200
6.7.1 Graphics View 体系结构	200
6.7.2 Graphics View 坐标系	201
6.7.3 深入 Graphics View	202
6.8 图形图像打印	208
6.8.1 普通打印过程	208
6.8.2 特殊窗口部件的打印	210
6.9 小结	211
第 7 章 拖放操作和剪贴板	212
7.1 拖放操作	212
7.1.1 拖放操作	212
7.1.2 定义新的拖放操作类型	214
7.1.3 Graphics View 框架下的拖放	215
7.2 使用剪贴板	217
7.3 小结	218
第 8 章 文件处理	219
8.1 读写文本文件	219
8.2 操作二进制文件	220
8.3 临时文件	222
8.4 目录操作和文件管理	222
8.4.1 目录操作	222
8.4.2 文件管理	224
8.5 监视文件系统变化	225

8.6 文件引擎	226
8.7 小结	226
第 9 章 网络	227
9.1 FTP 客户端	227
9.2 HTTP 客户端	235
9.3 UDP 应用	239
9.4 TCP 应用	243
9.5 高级应用	253
9.5.1 底层操作	253
9.5.2 使用代理	256
9.5.3 扩展 Qt 网络功能	256
9.5.4 效率问题	260
9.6 小结	260
第 10 章 多线程	261
10.1 启动一个线程	261
10.2 线程互斥与同步	264
10.2.1 临界区问题	265
10.2.2 使用 QMutex	265
10.2.3 使用 QSemaphore	266
10.2.4 使用 QWaitCondition	269
10.3 线程的其他问题	271
10.3.1 优先级问题	271
10.3.2 死锁及优先级反转问题	274
10.3.3 本地存储问题	275
10.4 Qt 的线程机制	276
10.4.1 可重入与线程安全	276
10.4.2 线程与事件循环	277
10.4.3 线程与信号/槽机制	278
10.4.4 多线程网络示例	279
10.5 小结	282
第 11 章 事件处理	283
11.1 事件机制	283
11.1.1 事件来源与类型	283
11.1.2 事件处理方法	284
11.2 事件处理器	285
11.3 事件过滤器	290
11.4 加快用户界面响应	292
11.4.1 使用 processEvents() 函数	293
11.4.2 使用定时器	294

11.5 小结	296
第 12 章 数据库	297
12.1 连接数据库	297
12.2 常用数据库操作	301
12.2.1 使用 SQL 语句	302
12.2.2 事务操作	304
12.2.3 使用 SQL 模型类	304
12.2.4 数据表示	308
12.3 Qt 数据库应用	310
12.3.1 使用嵌入式数据库	310
12.3.2 使用 Oracle 数据库	313
12.4 小结	325
第 13 章 Qt 的模板库和工具类	326
13.1 Qt 容器类	326
13.1.1 QList、QLinkedList 和 QVector	327
13.1.2 QMap、QHash	332
13.2 QString	334
13.2.1 隐式共享	335
13.2.2 内存分配策略	336
13.2.3 操作字符串	336
13.2.4 查询字符串数据	337
13.2.5 字符串的转换	338
13.3 QVariant	339
13.4 Qt 的算法	341
13.5 正则表达式	342
13.5.1 基本的正则表达式	342
13.5.2 文字捕获	344
13.6 小结	345

高级篇

第 14 章 XML	348
14.1 DOM	348
14.1.1 DOM 入门	348
14.1.2 使用 DOM	348
14.1.3 使用 DOM 写 XML 文件	352
14.2 SAX	354
14.3 基于流的 XML API	359
14.4 小结	365

第 15 章 模型/视图结构	366
15.1 模型/视图结构与 MVC 设计	
15.1 模式	366
15.1.1 模型	366
15.1.2 视图	367
15.1.3 代理	368
15.2 使用已有的模型视图类	368
15.2.1 使用已有的模型和视图类	368
15.2.2 QListWidget、QtreeWidget 和 QTableWidget	370
15.3 模型 (Models)	381
15.3.1 模型索引	381
15.3.2 模型角色	382
15.3.3 自定义模型	382
15.3.4 代理模型	385
15.4 视图 (Views)	390
15.4.1 自定义视图	390
15.4.2 数据-窗口部件映射	390
15.5 代理 (Delegates)	396
15.5.1 使用已有的代理	396
15.5.2 自定义代理	396
15.6 拖放与选中	401
15.6.1 拖放操作	401
15.6.2 选中模式	404
15.7 小结	405
第 16 章 高级绘图	406
16.1 3D 绘图——使用 OpenGL	406
16.1.1 创建 OpenGL 窗口	406
16.1.2 着色	410
16.1.3 3D 和旋转	411
16.1.4 纹理贴图	414
16.2 SVG	417
16.2.1 绘制 SVG 图形	418
16.2.2 生成 SVG 文件	419
16.3 小结	420
第 17 章 进程与进程间通信	421
17.1 使用 QProcess	421
17.2 Linux 进程间通信	423
17.3 新型进程间通信——D-Bus	425

17.3.1 D-Bus 简介	425	20.1.2 汉字编码	469
17.3.2 安装 QtDBus 模块	427	20.1.3 编码转换	469
17.3.3 接口与适配器	429	20.2 Qt Linguist	471
17.3.4 QtDBus 应用实例	432	20.2.1 发布管理器	472
17.4 小结	441	20.2.2 翻译器	474
第 18 章 Qt 插件	442	20.2.3 加载翻译文件	476
18.1 Qt 插件开发基础	442	20.3 语言切换	477
18.2 Qt 设计器插件	443	20.4 小结	477
18.2.1 使用 Scratchpad	443	第 21 章 Qt 单元测试框架	478
18.2.2 提升自定义窗口部件	444	21.1 QTestLib 框架	478
18.2.3 Qt 设计器插件开发	444	21.1.1 QTestLib	478
18.3 编写数据库插件	451	21.1.2 第一个 Qt 单元测试	478
18.4 自定义风格插件	455	21.2 数据驱动测试	480
18.5 小结	458	21.3 GUI 测试	481
第 19 章 脚本——QtScript	459	21.2.1 仿真 GUI 事件	481
19.1 执行 ECMAScript 脚本	459	21.2.2 重放 GUI 事件	483
19.2 QtScript 中的信号和槽	460	21.3 小结	484
19.3 使用 JavaScript 操作 Qt 对象	463	附录 A Qt 安装	485
19.4 基于 Prototype 的继承	467	附录 B Qt 集成开发环境	492
19.5 小结	467	附录 C qmake 速查	501
第 20 章 国际化	468	附录 D 深入 Qt 源代码	506
20.1 Unicode 与字符编码	468	附录 E Qt 资源	512
20.1.1 Unicode	468		



初 级 篇

第1章 Qt初步实践

第2章 对话框——QDialog

第3章 基础窗口部件——QWidget

第4章 程序主窗口——QMainWindow

第5章 布局管理

第 1 章 Qt 初步实践

作为 Qt 程序开发之旅的第一站，本章将阐述如何使用 Qt 开发一个简单的 GUI 用户界面程序。在这一章，将学习如何建立 Qt 主程序，建立 qmake 工程，还将接触到“信号（signal）”和“槽（slot）”以及“Qt 布局”等基本概念。随着学习的不断深入，将在第 3 章和第 5 章对这些概念进行深入的讲解，并演示它们在 GUI 用户界面设计中的应用。

1.1 第一个 Qt 程序

在这一节，学习创建第一个较简单的 Qt 应用程序。在这个程序中，用户界面将显示一行中文“同一个世界，同一个梦想！”通过这个程序，将学会使用两种手段建立 Qt 应用程序：KDevelop 集成开发环境和 vim 编辑器。

1.1.1 建立主程序

首先，看一下第一个 Qt GUI 应用程序 hello 的源代码，其内容如下所示（为了便于读者查阅相关源代码，代码的第一行注释了该文件在源代码中的路径）。

```
// chapter01/hello/src/hello.cpp.
#include <QtGui/QApplication>
#include <QtGui/QWidget>
#include <QtGui/QLabel>
#include <QtCore/QTextCodec>
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForName("gb18030"));
    QWidget* pWidget = new QWidget;
    QLabel label(pWidget);
    label.setText(QObject::tr("同一个世界，同一个梦想！"));
    pWidget->show();
    return app.exec();
}
```

这个程序的功能是在一个窗口中显示“同一个世界，同一个梦想！”，运行效果如图 1-1 所示。

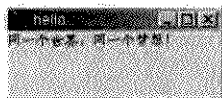


图 1-1 第一个 Qt 程序



扩展阅读

窗口 (Window) 和窗口部件 (Widget)

本书中,多次使用了窗口和窗口部件的概念。称一个图形用户界面为窗口,它往往具有标题栏、窗口边框 (frame),能够通过鼠标拖动和改变大小等特性,最典型的窗口就是对话框。例如,第一个 Qt 应用程序的用户界面就是一个窗口。当文中使用窗口的时候,就是特指这种情况。

一般的,窗口部件是对所有图形用户界面的统称,它既可以作为单独的窗口出现,也可以出现在一个窗口的内部。

1.1.2 建立工程

现在暂不分析第一个 Qt 应用程序是如何运行的,而是先为它建立 qmake 工程,然后进行调试和运行。

在 Linux 系统中,可以有多种方法输入、编辑上述 Qt 程序,此处将使用 vim 文本编辑器和 KDevelop 工具建立上述主程序。

在 vim 中建立 Qt 应用程序,步骤如下 (如果读者觉得在 vim 文本编辑器中编辑、调试以及运行 Qt 应用程序比较麻烦,可以直接跳过这一步,而选择在 KDevelop 集成开发环境中建立该 Qt 应用程序):

01 打开控制台程序 konsole,将当前目录切换到相应的路径下,执行控制台命令“mkdir hello”建立 hello 目录,执行“cd hello”命令进入该目录;

02 在控制台执行“vim hello.cpp”命令 (如果文件 hello.cpp 已经存在,则打开文件;否则新建 hello.cpp 文件),打开 vim 文本编辑器;

03 进入 vim 编辑器后按“i”键 (即打开 vim 编辑器的修改编辑功能),然后输入第一个 Qt 应用程序 hello 的源代码;

04 按“Esc”键,退出 vim 文本编辑器的编辑功能;

05 最后,在 vim 文本编辑器的命令行输入命令“:wq”,按回车键后 vim 将保存文件并退出。

现在,第一个 Qt 应用程序 hello 已经输入到 vim 文本编辑器中了。关于 vim 编辑器的使用读者可查阅相关帮助文档,在此不再赘述。

KDevelop 是集编辑、编译、调试和运行 C++ 程序等诸工具于一身的应用程序集成开发环境。在 KDevelop 中建立 Qt 应用程序并进行编译、调试、运行都是比较简单的,但需要建立 KDevelop 工程 (这和使用 VC++ 进行应用程序开发是类似的),对于初学者来说过程有些复杂。具体步骤如下 (注意,在 KDevelop 中输入汉字目前有些问题,解决办法见附录 B.1);

01 打开 KDevelop 后,选择菜单“工程”|“新建工程”,如图 1-2 所示。

02 在“建立新工程”对话框的“所有工程”选项卡中,选择“C++ | QMake project | Basic Qt 4 Application”,选择或者输入存放位置 (例如,“/home/lcf/book/chapter01”),输入应用程序名称“hello” (KDevelop 将会在/home/lcf/book/chapter01/路径下建立 hello 目录),单击“下一步”按钮,如图 1-3 所示。

03 设置“工程选项”,在此输入 Qt 4 的 qmake 和 Qt 设计器的绝对路径,直接单击“下一步”按钮,如图 1-4 所示。

04 设置“版本控制系统”,略过,单击“下一步”按钮,如图 1-5 所示。

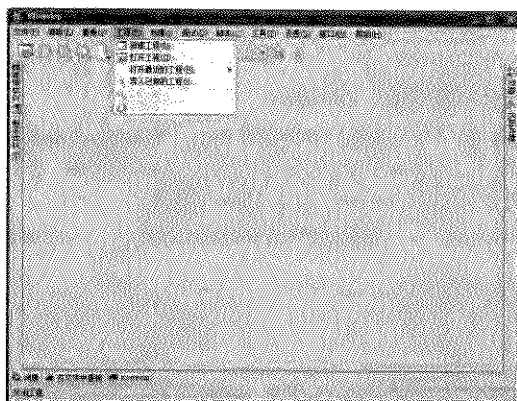


图 1-2 新建 KDevelop 工程

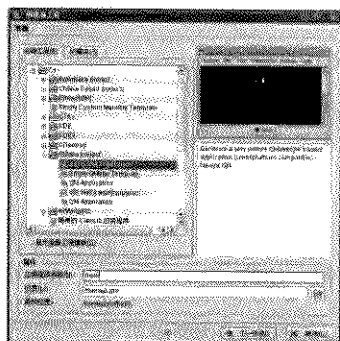


图 1-3 建立 hello 工程

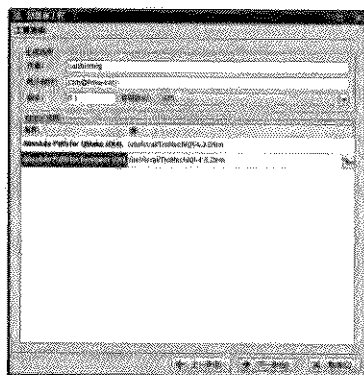


图 1-4 设置工程选项

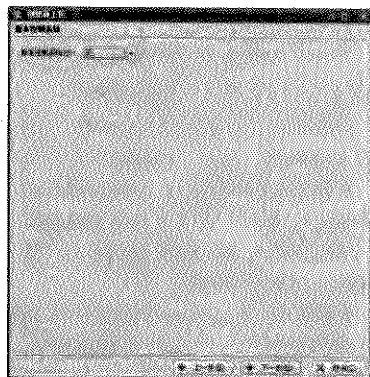


图 1-5 设置版本控制

05 在“h 文件的模板”选项（如图 1-6 所示）中，可以设置头文件 .h 的格式（在此省略）；单击“下一步”按钮进入“cpp 文件的模板”选项卡（如图 1-7 所示），与“h 文件的模板”类似。

06 最后，单击“完成”按钮，KDevelop 会自动生成一个标准的 C++ 主程序。在此，编辑修改为第一个 Qt 应用程序 hello 的源代码，如图 1-8 所示。

到此，在 KDevelop 中已经建立了一个 KDevelop 工程，并且输入了第一个 Qt 应用程序 hello。接下来，建立 Qt 应用程序的 qmake 工程文件。

有两种方法建立 qmake 工程：自动生成和手动建立。下面分别描述如何使用这两种方法建立应用程序 hello 的 qmake 工程。

1. 自动建立 qmake 工程

对于比较简单的小应用程序，使用 qmake 命令自动建立的 qmake 工程完全可以满足需要。

前面，已经在 vim 文本编辑器中输入了第一个 Qt 应用程序 hello 的源代码，现在为它建立相应的

qmake 工程。

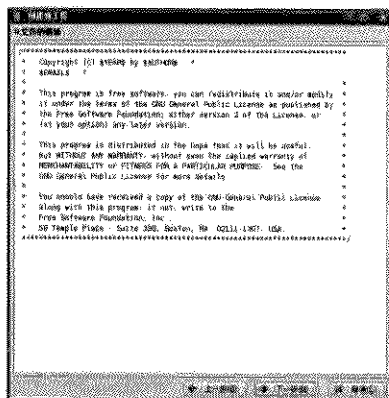


图 1-6 设置头文件模板

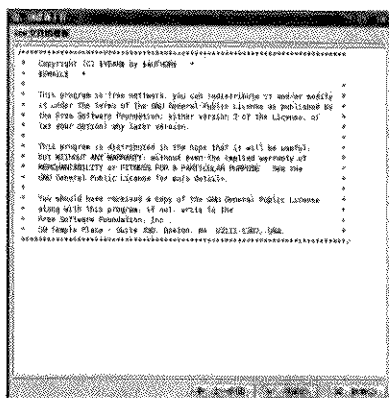


图 1-7 设置实现文件模板

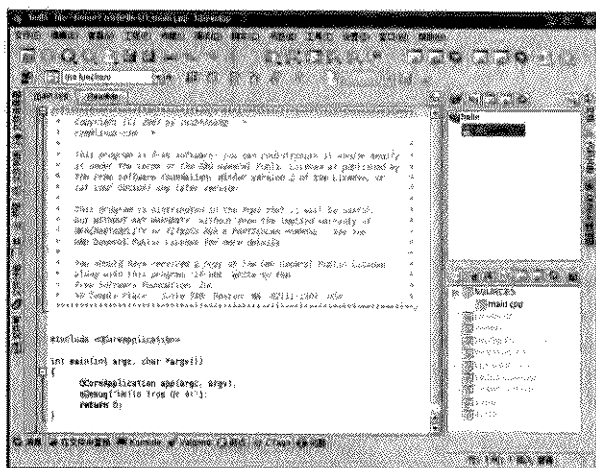


图 1-8 第一个 KDevelop 工程

首先，在控制台 `konsole` 中将当前目录切换到 `hello.cpp` 文件所在的目录，运行“`qmake -project`”命令。此时 Qt 的 `qmake` 工具将在当前目录下自动生成应用程序 `hello` 的工程文件 `hello.pro`，其内容如下。

```
#####
# Automatically generated by qmake (2.01a) ?? ?? 23 16:44:04 2007
#####
TEMPLATE = app
TARGET =
```

Input

下面分析一下这个 `qmake` 工程文件。

动生成的，以及文件生成的时间和 `qmake` 工具的版本号。

工具定义了5种模板:

- VC 应用程序 vcapp, 为 Visual Studio 生成一个应用程序工程, 仅仅用于 Windows 操作系统。

由于第一个 Qt 程序是一个可直接执行的应用程序，因此采用“应用程序 app”模板。

工程文件采用默认方式 (TARGET 的值为空), 即应用程序的名字采用工程文件 `hello.pro` 所在的文件来的名字 `hello`。

值设置为当前目录。

该值设置为当前目录。

件名字。应用程序 `hello` 比较简单，只包含一个 `hello.cpp` 源文件。

现在，在 KDevelop 中建立 qmake 工程文件。

应用程序同名的 `hello` 目录，它的主要目录结构如图 1-9 所示。

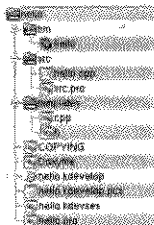


图 1-9 第一个 Qt 程序的目录结构

重刊

- src 目录存放源文件（包括 omake 工程文件 src.pro 以及资源文件等）；

- templates 目录存放 KDevelop 生成源文件.cpp 和头文件.h 时使用的模板;
- hello 目录下的其他文件都是关于 KDevelop 工程的 (hello.pro 是自动生成的 qmake 工程文件)。

可以看到, KDevelop 在上面的目录结构中生成了两个 qmake 工程文件: hello 目录下的 hello.pro 主工程文件和 hello/src 目录下的 src.pro 子工程文件。

主工程文件 hello.pro 的内容如下。

```
TEMPLATE=subdirs
SUBDIRS=src
```

第 1 行的 TEMPLATE 变量的值为 subdirs, 表示 hello.pro 工程在子文件夹中还包含子工程。子工程文件所在的目录由第 2 行 SUBDIRS 变量指定。在运行 qmake 生成 Makefile 文件的时候, qmake 工具会根据主工程 hello.pro 中的 SUBDIRS 选项自动到相应的目录下寻找 src.pro 子工程文件, qmake 工具将联合两个工程文件分别生成两个前后相关的 Makefile 文件。

qmake 子工程文件 src.pro 的内容如下。

```
SOURCES=hello.cpp
TARGET=../bin/hello
```

该工程文件中, 大多数变量 (或选项) 采用了默认值, 仅仅定义了 SOURCES 和 TARGET 工程选项, 在此不再详述。

注意, 工程文件中的源文件.cpp 和头文件.h 的位置, 既可以采用绝对路径表示, 也可以采用相对路径表示。相对路径意味着源文件或头文件相对于工程文件.pro 所在目录的路径。例如, 在第一个 Qt 应用程序的例子中, src.pro 工程文件在 /hello/src 目录下, hello.cpp 源文件也在 /hello/src 目录下。那么采用相对路径时, 工程文件 src.pro 的选项 SOURCES 的值为

./hello.cpp (或 hello.cp)

而采用绝对路径时, 则值为

<KDevelop 工程所在的绝对路径>/src/hello.cpp

其他工程选项也是类似的, 比如 src.pro 工程文件中的 TARGET 变量值为 ../bin/hello, 表示目标文件将被放置在 <KDevelop 工程所在的绝对路径>/bin 目录下。

2. 手动建立 qmake 工程

建立较大型 Qt 应用程序时, 使用 qmake 工具生成的 qmake 工程文件无法满足编程的需要。尽管 qmake 工具生成的工程文件完全可以满足 hello 程序的编译需求, 但作为一个例子, 笔者生成了对应于 vim 输入的源代码文件的一个较简单的 hello.pro 文件, 它位于 hello 目录下。其内容如下:

```
TEMPLATE = app
TARGET = hello
DESTDIR = .
CONFIG += debug \
        warn_on
OBJECTS_DIR= ./tmp
SOURCES += hello.cpp
```

下面, 简单地分析一下这个工程文件。

- 变量 TARGET 定义了可执行目标文件的名字为 hello。

- 变量 DESTDIR 定义了存放可执行目标文件 hello 的路径，即在 hello 目录下。
- 变量 CONFIG 定义了编译选项，即：
 - debug 表示建立的目标代码是调试版本（相对于 release 发布版本）；
 - warn_on 要求编译器在编译应用程序时打开警告开关。
- 变量 OBJECTS_DIR 描述了编译 / 连接应用程序过程中产生的中间文件存放的位置。即将编译器生成的中间文件 hello.o 放置在工程文件所在目录的 tmp 子目录下（hello/tmp）。在为应用程序 hello 建立 qmake 工程后，就可以编译运行第一个 Qt 应用程序了。

1.1.3 编译/运行第一个 Qt 应用程序

下面，通过两种方式编译、运行第一个 Qt 应用程序。

在控制台模式下，依次在工程文件所在的目录下执行下述命令：

- qmake，生成 Makefile 文件；
- make，编译、连接并生成可执行代码；
- ./hello，执行应用程序。

在 KDevelop 中，编译、运行 Qt 应用程序的步骤如下：

01 在右侧“QMake 管理器”选项卡的“src”子工程下单击右键，选择菜单命令“运行 qmake”，建立 Makefile 文件，如图 1-10 所示。

02 选择“编译”菜单中的“编译工程”命令（或单击工具栏中的“编辑工程”工具按钮），编译、连接 Qt 程序，如图 1-11 所示。



图 1-10 KDevelop 中运行 qmake



图 1-11 执行编辑工程命令

03 选择“编译”菜单中的“执行主程序”命令（或单击工具栏中的“执行主程序”工具按钮），运行 hello 程序。应用程序的运行效果如图 1-12 所示。



图 1-12 第一个 Qt 程序运行效果

1.1.4 第一个 Qt 程序的代码分析

现在再回过头来逐行地分析一下源代码，看看第一个 Qt 应用程序是如何运行的。

这是一个非常简单的 Qt 应用程序，它不包含任何自定义的类，且仅仅有一个 `hello.cpp` 主程序文件。

```
// chapter01/hello/src/hello.cpp.
#include <QtGui/QApplication>
#include <QtGui/QWidget>
#include <QtGui/QLabel>
#include <QtCore/QTextCodec>
```

首先包含必要的 Qt 类的头文件。

`#include <QtGui/QApplication>` 表示包含 Qt 的应用程序类 `QApplication` 的头文件，其中的 `QtGui` 表示 Qt 的 `QtGui` 模块，从目录结构来讲，`QtGui` 是文件夹。`<QApplication>` 是 Qt 定义 `QApplication` 类的头文件（与类名相同）；`QApplication` 类是每一个 Qt GUI 应用程序所必需的。

`#include <QtGui/QWidget>` 将 Qt 的基础窗口部件 `QWidget` 的定义包含进来；接下来的 `#include <QtGui/QLabel>` 将 Qt 的标签 `QLabel` 头文件包含进来。事实上，`QLabel` 头文件中已经包含了 `QWidget` 的定义，因此也可以去掉对 `QWidget` 头文件的包含。

`#include <QtCore/QTextCodec>` 用于包含 `QtCore` 模块下的 `QTextCodec` 类的头文件。类 `QTextCodec` 封装定义了显示文本（例如，“同一个世界，同一个梦想”）字符集的转化功能。



扩展阅读

Qt 4 定义了多个模块，每个模块包含相对独立的库文件并实现各自相应的功能。Qt4 的主要模块有：

- `QtCore`，Qt 4 的基本模块，定义了其他模块使用的 Qt 核心的非 GUI 类，所有其他的模块都依赖于该模块；
- `QtGui`，定义了图形用户界面类；
- `QtNetwork`，定义了 Qt 的网络编程类；
- `QtOpenGL`，定义了 OpenGL 的支持类；
- `QtSql`，定义了访问数据库的类；
- `QtSvg`，定义了显示和生成 SVG（Scalable Vector Graphics）类；
- `QtXml`，定义了处理 XML（eXtensible Markup Language）语言的类；
- `QtDesigner`，定义了扩展 Qt 设计器（Qt Designer）的类，该模块使得程序员能够为 Qt 设计器创建自定义的 Qt 窗口部件插件（widget plugins）和创建能够访问 Qt 设计器组件的类；
- `QtUiTools`，定义了应用程序中直接处理 ui（User Interface）文件的类，它使得应用程序能够在运行时使用 ui 文件构建用户界面；
- `QtAssistant`，为应用程序提供了加载 Qt 助手（Qt Assistant）以支持在线帮助（online help）的功能；
- `Qt3Support`，定义了同 Qt4 以前版本 Qt3 兼容的类，以使得 Qt3 的程序能够更容易地移植到 Qt4；
- `QtTest`，定义了对 Qt 应用程序和库进行单元测试（unit testing）的类。

UNIX 平台的 Qt4 版本还包含 `QtDBus` 扩展模块，该模块提供了使用 D-Bus 进行进程间通信（Inter-Process Communication, IPC）的 Qt 类。

最新版的 Qt 4.3 增加了一个新模块 `QtScript`，该模块提供了对脚本的支持。

此外，Windows 平台的 Qt 商业版还包含两个扩展模块：

- （1）`QAxContainer`，定义了访问 ActiveX 控件和 COM（Component Object Model）对象的扩展；

(2) QXServer, 一个静态库, 用于将一个标准的 Qt 二进制代码转化为 COM 服务器 (COM server)。

在 qmake 工程中, 默认情况下已经包含了 QtCore 和 QtGui 模块 (如果不想使用 QtGui 模块, 而仅仅使用 QtCore 连接程序, 可以在 qmake 工程文件中通过使用 “QT = gui” 来取消对 QtGui 模块的包含), 因此无需配置就可以使用这两个模块中的类。而对于 Qt 的其他模块, 在使用之前必须在 qmake 工程文件中通过 QT 选项进行配置 (将在各个章节中详细阐述)。

一般可以在应用程序中通过 #include <QtGui/QtGui> 包含整个 QtGui 模块所有类的头文件, 其中第一个 QtGui 是模块名, 第二个 QtGui 是 QtGui 模块 (文件夹) 下的预定义头文件 (或者使用 #include <QtGui>, 其效果相同, 不过此时 <QtGui> 是 QtGui 模块 (文件夹) 下的预定义头文件); 也可以单独包含某个类的头文件: #include <QtGui/QApplication> (或者 #include <QApplication>)。

```
int main ( int argc, char* argv[] )
{
    QApplication app( argc, argv );
```

创建一个 QApplication 对象并将用户在控制台输入的参数传递给该应用程序对象。QApplication 对象管理 Qt GUI 应用程序的控制流程和主要的设置选项。使用 Qt 设计的任何 GUI 应用程序, 都必须包含一个 QApplication 对象。而对于非 GUI 的 Qt 应用程序, 可以使用不依赖于 QtGui 库的 QCoreApplication。

```
QTextCodec::setCodecForTr( QTextCodec::codecForName("gb18030"));
```

该行代码设置 QObject::tr() 使用的字符集。在第一个 Qt 应用程序中, 显示的是中文字符 “同一个世界, 同一个梦想!”, 因此程序采用的是字符集 “GB18030” (GB18030-2000 是取代 GBK1.0 的正式国家标准。现在的 PC 平台必须支持 GB18030)。如果不采用正确的字符集, Qt 应用程序中的中文字符将显示为乱码。如果仅仅显示英文字符的话, 可以不使用该行代码, 并且将下面的设置标签文本的代码修改为 label.setText(“one world, one dream.”)。

```
QWidget* pWidget = new QWidget;
```

创建一个 QWidget 对象。

```
QLabel label( pWidget );
label.setText( QObject::tr( “同一个世界, 同一个梦想! ” ) );
```

创建一个 QLabel 对象, 并将该标签的父窗口部件设置为 pWidget, 这将使得 QLabel 对象放置在 QWidget 界面上。QLabel::setText() 函数设置 QLabel 对象的显示文本为 “同一个世界, 同一个梦想!”。

```
pWidget->show();
return app.exec();
}
```

函数 QWidget::show() 将创建的图形用户界面呈现在显示器上。

最后程序返回 Qt 应用程序对象 app 执行的结果, 并退出。

QApplication::exec() 语句的执行, 将使得 Qt GUI 进入一个主事件循环, 直到程序中调用 exit()、quit() 或关闭应用程序的主窗口。主事件循环开始后, 它将会接收用户界面事件以及其他事件源的事件并向相应的窗口进行分发和处理。此外, 它还完成 Qt 应用程序的初始化和应用程序运行结束后的善后处理, 并提供会话管理 (session management)。

在 Linux 平台下，编译应用程序时，编译器经常会给出“no newline at end of file.”的警告。它的意思是说，在应用程序代码文件的最后没有新行。为了消除这个编译器警告，必须在应用程序代码的最后加一行没有任何字符（也不能包含任何的空格字符）的空行。

1.2 使用 Qt 布局管理器

在第一个 Qt 程序中，程序在窗口上直接加载了一个 QLabel 窗口部件，以显示一些文本信息。然而，在较复杂的 GUI 用户界面上，仅仅通过指定窗口部件的父子关系以期达到加载和排列窗口部件的方法是行不通的，最好的办法是使用 Qt 提供的布局管理器。

在第一个 Qt GUI 应用程序的基础上，现在加入一个“关闭”按钮，并学习如何使用 Qt 的布局管理器进行界面的排布和外观控制。

首先，看一下新的应用程序的实现代码。

```
#include <QtGui/QApplication>
#include <QtGui/QWidget>
#include <QtGui/QLabel>
#include <QtCore/QTextCodec>
#include <QtGui/QPushButton>
#include <QtGui/QVBoxLayout>
```

将用到的按钮类 QPushButton 和布局管理器类 QVBoxLayout 的头文件包含进来。

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForName("gb18030"));

    QWidget* pWidget = new QWidget;
    QLabel label(pWidget);
    label.setText(QObject::tr("同一个世界，同一个梦想！"));
    QPushButton* btn = new QPushButton(QObject::tr("关闭"), pWidget);
```

创建一个名为“关闭”的 QPushButton 按钮对象，父窗口部件设置为 pWidget。

```
QVBoxLayout* layout = new QVBoxLayout;
layout->addWidget(&label);
layout->addWidget(btn);
pWidget->setLayout(layout);
```

创建一个垂直布局管理器 QVBoxLayout 对象 layout；函数 QVBoxLayout::addWidget() 将标签对象 label 和按钮对象 btn 放置在该管理器中；最后函数 QWidget::setLayout() 将垂直布局管理器 layout 添加到窗口部件对象 pWidget 中。垂直布局管理器 QVBoxLayout 将上述添加的标签 label 和按钮 btn 由上到下依次放置在 pWidget 用户界面中。

```
pWidget->show();
return app.exec();
}
```



最后显示窗口，并进入 Qt 应用程序事件循环。
编译，运行应用程序，运行界面如图 1-13 所示。

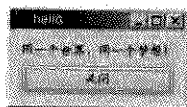


图 1-13 hello 程序的新面孔

1.3 关联操作

在 Qt 应用程序的用户界面加入“关闭”按钮后，应用程序并没有响应关闭操作。这是因为程序还没有将相应的信号和槽关联起来。

Qt 提供了信号和槽机制来完成界面操作的响应。因此，为了响应用户的关闭操作，需要将“关闭”按钮发送的单击信号 `QPushButton::clicked()` 和窗口部件 `QWidget` 的 `QWidget::close()` 槽关联起来。关联信号和槽的代码如下（省略了包含头文件部分）。

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForName("gb18030"));
    QWidget* pWidget = new QWidget;
    QLabel label(pWidget);
    label.setText(QObject::tr("同一个世界，同一个梦想！"));
    QPushButton* btn = new QPushButton(QObject::tr("关闭"), pWidget);
    QVBoxLayout* layout = new QVBoxLayout;
    layout->addWidget(&label);
    layout->addWidget(btn);
    pWidget->setLayout(layout);

    QObject::connect(btn, SIGNAL(clicked()), pWidget, SLOT(close()));

    pWidget->show();
    return app.exec();
}
```

函数 `QObject::connect()` 将“关闭”按钮 `btn` 的 `clicked()` 信号同窗口 `pWidget` 的 `close()` 槽关联起来。`QObject::connect()` 函数中，实参 `btn` 是发信号的源对象指针，实参 `pWidget` 是接受信号的目标对象指针；`clicked()` 信号和 `close()` 槽已经分别由 Qt 的 `QPushButton` 类和 `QWidget` 类定义了，所以无需事先声明和定义就可以直接使用；`SIGNAL()` 和 `SLOT()` 是 Qt 定义的两个宏，它们返回其参数的 C 风格字符串（`const char *`）。

此处，程序使用了 `QObject` 对象的静态函数 `connect()`，此外 `QObject` 类还提供了非静态形式的 `connect()` 函数。继承自 `QObject` 的 Qt 类都具有支持信号和槽的能力，并且可以在子类的实现代码中直接使用 `connect()` 函数。

关于 Qt 信号和槽机制的详细描述见第 3 章。在此，只是简单的介绍一下基本的知识。

现在，重新编译、连接并运行 `hello` 应用程序。单击“关闭”按钮，OK！应用程序立刻消失得无影无踪了。

1.4 小 结

在这一章，编写了一个很简单的 Qt 应用程序，学习了如何在 vim 文本编辑器和 KDevelop 集成开发环境中建立一个 Qt 应用程序的基本步骤；介绍了 Qt 布局管理器的基本使用方法，以及如何将 Qt GUI 窗口部件产生的信号和相关的槽进行关联和挂接。

通过这一章的学习，对使用标准 C++ 和 Qt 进行 GUI 应用程序开发有了一个基本的认识。下一章，将学习 Qt 的对话框——QDialog，学习如何继承 Qt 的窗口部件类，实现自定义对话框。

第2章 对话框——QDialog

对话框是一种特殊的窗口，它一般用来提供反馈信息或从用户处获取输入。对话框有各种各样的形状和尺寸，其范围从简单用于显示单行信息的对话框到包含复杂控件的大型对话框。

对话框给用户提供了一种同应用程序进行交互的便捷方式。因此,较大型的 GUI 应用程序中少不了形形色色的对话框,像系统选项的设置、字体的设置、风格的设置以及各种形式的向导等。这一章将学习如何在 Qt GUI 应用程序中创建和使用对话框,了解 Qt 的内建(built-in)对话框。

2.1 自定义对话框

QDialog 是所有 Qt 对话框窗口的基类，它继承自 QWidget，如图 2-1 所示。

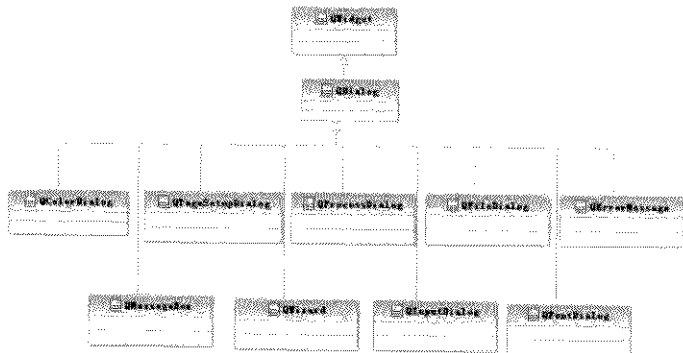


图 2-1 对话框类图

下面，根据 Qt 提供的 `QDialog` 基类建立一个自定义的对话框类。通过这个例子，将学习如何继承 `QDialog` 实现自定义的对话框，如何添加新的窗口部件，以及如何添加自己的界面控制代码。

在第 1 章中，使用了两种工具建立 Qt 应用程序。在接下来的章节中，将使用 KDevelop 集成开发工具进行代码的编辑和调试。笔者也推荐读者使用 KDevelop。

2.1.1 建立新类

首先，建立名字为“mydialog”的 KDevelop 工程。然后，为工程新建文件，步骤如下。

01 选择“文件”菜单中的“新建”命令，打开“新建文件”对话框，如图 2-2 所示：

02 在“新建文件”对话框中输入新建文件的名称“logindlg.h”，选中“添加到工程中”选项，然后单击“确定”按钮。

这时，新建的 `logindlg.h` 将会添加到工程中。打开 KDevelop 界面右侧的“QMake 管理器”选项卡，将会在它的“HEADERS”列表中看到这个文件，如图 2-3 所示。

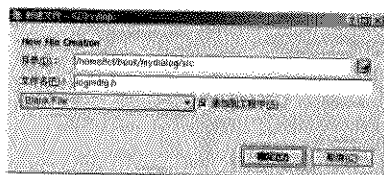


图 2-2 新建文件 logindlg.h

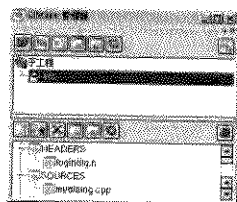


图 2-3 新建文件在工程中

使用同样的方法，在工程中加入“`logindlg.cpp`”文件。

自定义对话框类 `CLoginDlg` 的定义文件 `logindlg.h` 内容如下。

```
// chapter02/mydialog/src/logindlg.h
#ifndef _LOGINDLG_H_
#define _LOGINDLG_H_
```

定义宏变量，确保该头文件只被包含一次，防止头文件被多次包含。

```
#include <QtGui/QDialog>
class CLoginDlg : public QDialog
{
    Q_OBJECT
public:
    CLoginDlg(QWidget* = 0);
};
#endif
```

自定义类 `CLoginDlg` 继承自 `QDialog`，因此必须包含基类 `QDialog` 的定义。

`Q_OBJECT` 宏的作用是启动 Qt 元对象系统的一些特性（比如支持信号和槽等），它必须放置到类定义的私有区。Qt 元对象系统将在第 3 章介绍。

`CLoginDlg(QWidget* = 0)` 是构造函数，它指定了一个默认值为 `NULL` 的指向 `QWidget` 的参数。该形参定义了自定义对话框对象的父窗口部件，默认值 `NULL` 意味着自定义的对话框没有父窗口部件。

目前，`CLoginDlg` 类的文件 `dialog.cpp` 仅仅实现一个空白的构造函数。

```
// chapter02/mydialog/src/logindlg.cpp
#include "logindlg.h"
CLocalDlg::CLocalDlg(QWidget* parent)
: QDialog(parent)
{
}
```

编译、连接并运行程序。此时用户界面是一个没有任何内容的空白对话框。

2.1.2 添加子窗口部件

上述定义的对话框是一个空白对话框，现在加入一些新的窗口部件。自定义对话框的界面布局如



图 2-4 所示。

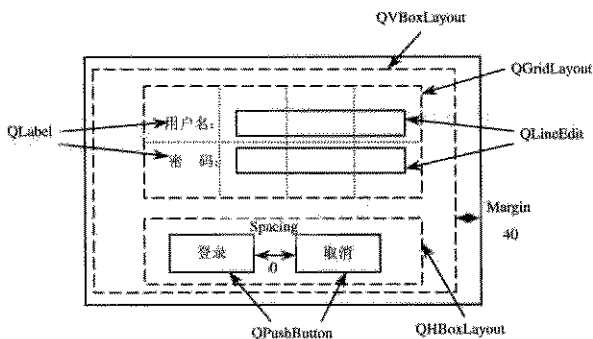


图 2-4 登录对话框界面布局

现在重新定义该类，其头文件 loginDlg.h 内容如下所示。

```
//chapter02/mydialog/src/loginDlg.h
#ifndef _LOGINDLG_H_
#define _LOGINDLG_H_
#include <QtGui/QDialog>
class QLineEdit;
class CLoginDlg : public QDialog
{
    Q_OBJECT
public:
    CLoginDlg(QWidget* = 0);
public slots:
    virtual void accept();
private:
    QLineEdit* usrLineEdit;
    QLineEdit* pwdLineEdit;
};
#endif
```

代码“class QLineEdit”是类 QLineEdit 的传递声明，因为在类 CLoginDlg 的头文件中仅仅使用了指向 QLineEdit 对象的指针，因此在编译头文件时，gcc 编译器不需要知道 QLineEdit 类的定义。该行代码的作用是告诉编译器，QLineEdit 类已经存在。这样做有如下好处：

- 首先，它减小了头文件的大小，增加了编译速度（特别是当该头文件被其他文件多次包含引用时）；
- 其次，这样做可以避免因包含头文件的顺序不当而造成连接错误，特别是在大的工程当中更应该避免随意地在一个头文件中包含另一个头文件。

在头文件中，重新声明了基类 QDialog 类的虚函数 accept()。在类 CLoginDlg 的实现文件 loginDlg.cpp 中，将重写该函数，目的是为了验证用户输入的用户名和密码的有效性。

在类的私有区，声明了指向 QLineEdit 对象的指针成员。其中，usrLineEdit 对象存放、显示用户输入的用户名，pwdLineEdit 对象存放、显示用户输入的密码。

现在来看一下自定义对话框类 CLoginDlg 的实现文件 loginDlg.cpp。

```
//chapter02/mydialog/src/loginDlg.cpp
#include <QtGui/QtGui>
#include "loginDlg.h"
```

在文件开头,包含了 Qt 用户界面头文件 QtGui。QtGui 头文件包含了 QtCore 模块和 QtGui 模块的所有 Qt 类的定义。由于在类 CLoginDlg 的实现文件 loginDlg.cpp 中用到了很多的 QtCore 类和 QtGui 类,通过包含 QtGui 头文件,可以避免——包含所有要使用到的 Qt 类的头文件。

下面是 CLoginDlg 类的构造函数,该函数完成登录对话框的初始化操作。

```
CLoginDlg::CLoginDlg(QWidget* parent)
: QDialog(parent)
{
```

调用 CLoginDlg 的父类 QDialog 的构造函数,并将实参 parent 传递给父类的构造函数,以设置登录对话框的父窗口部件。

```
QLabel* usrLabel = new QLabel(tr("用户名:"));
QLabel* pwdLabel = new QLabel(tr("密码:"));
usrLineEdit = new QLineEdit;
pwdLineEdit = new QLineEdit;
pwdLineEdit->setEchoMode(QLineEdit::Password);
```

创建 QLabel 标签对象 usrLabel 和 pwdLabel,以提示用户输入“用户名”和“密码”。

接下来,创建输入用户名及其密码的行编辑框 QLineEdit 对象 usrLineEdit 和 pwdLineEdit;

函数 QLineEdit::setEchoMode() 设置密码编辑框对象 pwdLineEdit 的内容显示方式为 QLineEdit::Password,即采用星号“*”代替用户输入的字符。通常行编辑框 QLineEdit 窗口部件的显示方式有下列几种方式:

- QLineEdit::Normal, 默认的显示方式,显示用户实际输入的内容;
- QLineEdit::Password, 用星号“*”代替用户实际输入的内容;
- QLineEdit::NoEcho, 不显示用户输入的任何内容,尽管 QLineEdit::Password 可以有效地起到密码保护的作用,但它仍然显示了用户输入字符的个数;而 QLineEdit::NoEcho 对显示内容采取了更进一步的保密措施;
- QLineEdit::PasswordEchoOnEdit, 仅仅用户在行编辑框里编辑文本的内容时,才显示用户输入的字符,而在用户完成编辑后以星号“*”代替输入的内容。

```
QGridLayout* gridLayout = new QGridLayout;
gridLayout->addWidget(usrLabel, 0, 0, 1, 1);
gridLayout->addWidget(usrLineEdit, 0, 1, 1, 3);
gridLayout->addWidget(pwdLabel, 1, 0, 1, 1);
gridLayout->addWidget(pwdLineEdit, 1, 1, 1, 3);
```

创建一个网格布局管理器 QGridLayout 对象 gridLayout,并将窗口部件添加到该布局管理器中。

函数 QGridLayout::addWidget() 将先前生成的标签和行编辑框添加到网格管理器 gridLayout 中。在上面的代码中, QGridLayout::addWidget(usrLineEdit, 0, 1, 1, 3) 函数共有 5 个参数,实参 usrLineEdit 指出哪一个窗口部件将被放置在网格布局管理器 gridLayout 中;后 4 个参数确定了行编辑框对象

usrLineEdit 在网格布局管理器 `gridLayout` 中的具体位置。其中前两个实参分别表示行和列的位置（行号和列号），后两个参数分别表示行的跨度和列的跨度，如图 2-5 所示。

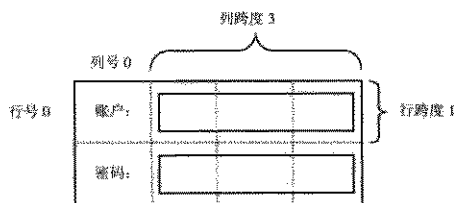


图 2-5 网格布局管理器的布局管理

```
QPushButton* okBtn = new QPushButton(tr("确定"));
QPushButton* cancelBtn = new QPushButton(tr("取消"));
QHBoxLayout* btnLayout = new QHBoxLayout;
btnLayout->setSpacing(60);
btnLayout->addWidget(okBtn);
btnLayout->addWidget(cancelBtn);
```

创建“确定”和“取消”两个 `QPushButton` 按钮对象。其中，`okBtn` 按钮完成对话框的登录验证；`cancelBtn` 按钮完成取消登录并退出对话框的功能。

接下来，创建排布和管理按钮窗口部件的水平布局管理器 `QHBoxLayout` 对象 `btnLayout`。

函数 `QHBoxLayout::setSpacing()` 设置水平布局管理器 `btnLayout` 对象内部窗口部件之间的间隔（如图 2-4 所示）为 60。

函数 `QHBoxLayout::addWidget()` 将 `okBtn` 按钮和 `cancelBtn` 按钮加入到水平布局管理器 `btnLayout` 中。这两个按钮将按放置的先后顺序依次从左向右排布（默认值），且之间的间隔为 60。

```
QVBoxLayout* dlgLayout = new QVBoxLayout;
dlgLayout->setMargin(40);
dlgLayout->addLayout(gridLayout);
dlgLayout->addStretch(40);
dlgLayout->addLayout(btnLayout);
setLayout(dlgLayout);
```

创建一个垂直布局管理器 `QVBoxLayout` 对象 `dlgLayout`。该管理器将排布和管理先前创建的所有布局管理器，进而完成对所有子窗口部件的管理。

函数 `QVBoxLayout::addLayout()` 将布局管理器 `gridLayout` 和 `btnLayout` 添加到 `dlgLayout` 布局管理器中。这样，`dlgLayout` 管理 `gridLayout` 和 `btnLayout`。而后两个管理器进一步管理放在它们当中的子窗口部件。

函数 `QVBoxLayout::setMargin()` 设置布局管理器 `dlgLayout` 边框的宽度为 40，即其内部子窗口部件距离布局管理器边界（布局管理器的边界是不可见的）的距离（如图 2-4 所示）为 40。

函数 `QVBoxLayout::addStretch()` 函数在垂直布局管理器 `dlgLayout` 对象中加入一个大小为 40 的 stretch，这将使得布局管理器 `gridLayout` 和 `btnLayout` 之间的默认距离设置为 40，同时当上下拉伸对话框的高度时，该 stretch 可以自由的伸缩，从而保证 `gridLayout` 和 `btnLayout` 管理器内部各窗口部件的高度以及彼此间的垂直距离保持不变。

函数 `QWidget::setLayout()` 函数将垂直布局管理器 `dlgLayout` 设置为登录对话框的顶层布局管理器。此时, 当任意拉伸对话框时, `dlgLayout` 布局管理器都能够有效地管理对话框内部的所有子窗口部件。

```
connect(okBtn, SIGNAL(clicked()), this, SLOT(accept()));
connect(cancelBtn, SIGNAL(clicked()), this, SLOT(reject()));
```

关联“确定”按钮的 `QPushButton::clicked()` 信号到登录对话框的 `accept()` 槽, 当用户单击“确定”按钮时能够进行用户名和密码的验证。

接下来, 关联“取消”按钮的 `QPushButton::clicked()` 信号到登录对话框的 `reject()` 槽, 以使得对话框能够响应用户的退出操作。 `QDialog::reject()` 槽将会隐藏登录对话框, 并将对话框的返回代码设置为 `QDialog::Rejected`。此时对话框将关闭, 启动对话框的槽函数 `QDialog::exec()` 将返回运行结果 `QDialog::Rejected`。

```
setWindowTitle(tr("登录"));
resize(300, 200);
}
```

函数 `QWidget::setWindowTitle()` 设置对话框的标题为“登录”。

最后, 函数 `QWidget::resize()` 重新设置对话框的大小为 (300, 200), 即高为 300、宽为 200。

为了实现对“用户名”和“密码”的有效性验证, 本例重写了 `QDialog` 的虚槽函数 `accept()`, 该函数响应用户单击“确定”按钮的 `QPushButton::clicked()` 信号。

```
void CLoginDlg::accept()
{
    if(usrLineEdit->text().trimmed() == tr("lcf")
        && pwdLineEdit->text() == tr("lcf"))
    {
        QDialog::accept();
    }
}
```

取出用户输入的用户名和密码并和预定的值进行比对, 如果用户名和密码全部正确, 调用父类的 `QDialog::accept()` 槽函数, 该函数将关闭模态对话框, 设置对话框的运行结果为 `QDialog::Accepted`, 并发送 `QDialog::finished(int result)` 信号。此时对话框将关闭, 启动对话框的槽函数 `QDialog::exec()` 将返回运行结果 `QDialog::Accepted`。

函数 `QString::text()` 返回行编辑框中的文本内容, 返回的类型是一个 `QString` 字符串。函数 `QString::trimmed()` 移除在字符串开头和结尾的空白字符, 并返回移除空白字符后的字符串。这些空白字符包括 ASCII 字符 “\”、“\n”、“\r”、“\f”、“\v” 和空格 “ ”。

```
else
{
    QMessageBox::warning(this,
        tr("警告"),
        tr("用户名或密码错误!"),
        QMessageBox::Yes);
    usrLineEdit->setFocus();
}
}
```

如果用户输入的用户名或密码有一个错误, 那么程序将执行上面这段代码。它将显示一个警告提示框。



告诉用户其输入的用户名和密码是有错误的。

函数 `QMessageBox::warning()` 将会创建并显示一个模态的警告对话框，提示用户输入的用户名或密码错误。

函数 `QLineEdit::setFocus()` 将鼠标的焦点定位到编辑框对象 `usrLineEdit`，以方便用户进行“用户名”或“密码”的修改。此段代码运行后，登录对话框将返回到可输入/操作的状态，此时用户可以进行相关的操作，或修改用户名/密码，或选择退出。

`QMessageBox` 类提供了显示操作信息的几种模态对话框：

- `QMessageBox::about`，一个仅仅带有标题和简单文本的消息框（如图 2-6 所示），一般用于显示帮助提示信息。
- `QMessageBox::aboutQt`，显示关于 Qt 的消息框（如图 2-7 所示），包括 Qt 版本以及 Trolltech 公司的产品信息等。

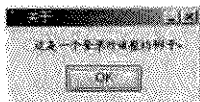


图 2-6 关于消息框

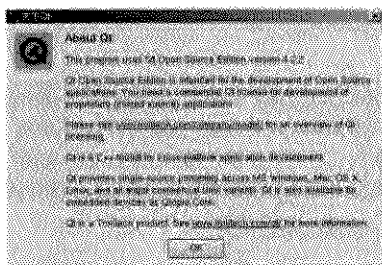


图 2-7 关于 Qt 消息框

- `QMessageBox::information`，一个具有主题和提示文本的提示消息框（如图 2-8 所示），程序员可以根据情况定制按钮的个数，以及各个按钮的角色。
- `QMessageBox::question`，一个具有标题和文本信息的询问消息框（如图 2-9 所示），程序员可以根据情况定制按钮的个数，以及各个按钮的角色。



图 2-8 提示消息框



图 2-9 询问消息框

- `QMessageBox::warning`，一个具有标题和文本信息的警告消息框（如图 2-10 所示），程序员可以根据情况定制按钮的个数，以及各个按钮的角色。
- `QMessageBox::critical`，一个具有标题和文本信息的致命错误消息框（如图 2-11 所示），程序员可以定制按钮的个数，以及各个按钮的角色。



图 2-10 警告消息框



图 2-11 致命错误消息框

QMessageBox 的几种模态消息框的标题栏 (title bar) 使用父窗口部件的图标; 此外, QMessageBox 提供的后四种消息框都具有返回值, 程序可以根据用户的响应情况进行下一步的处理。下面看一个消息框类 QMessageBox 高级应用的例子:

```
QMessageBox box;
box.setWindowTitle(tr("警告"));
box.setIcon(QMessageBox::Warning);
box.setText(tr("程序安装错误, 是否退出?"));
box.setStandardButtons(QMessageBox::Yes
                       | QMessageBox::No);
box.setDetailedText(tr("请查看安装介质有无损坏。"));
switch(box.exec())
{
    case QMessageBox::Yes:
        // 进行下一步处理
        break;
    case QMessageBox::No:
        // 进行下一步处理
        break;
    default:
        // 进行默认处理
        break;
}
```

第1行代码调用 QMessageBox 的默认构造函数创建一个 QMessageBox 栈对象 box, 父窗口部件设置为 NULL。

函数 QMessageBox::setIcon() 设置消息框的图标为 QMessageBox::Warning, 即消息框是一个警告消息框。

函数 QMessageBox::setWindowTitle() 设置消息框的标题为“警告”。

函数 QMessageBox::setStandardButtons() 设置消息框的标准按钮 (也可以调用 QMessageBox::addButton() 函数添加自定义的按钮, 并通过 QMessageBox::clickedButton() 函数获取用户单击的按钮), 此处添加了两个标准按钮: QMessageBox::Yes 和 QMessageBox::No。

函数 QMessageBox::setDetailedText() 添加“Show Details...”按钮, 详细信息内容设置为“请查看安装介质有无损坏。”。

消息框的运行效果如图 2-12 所示。

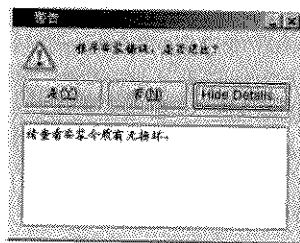


图 2-12 一个复杂的消息框



2.2 加入主程序

目前,已经编写了登录对话框类的头文件及其实现代码,现在把它加入到主程序中编译运行。

```
// chapter02/mydialog/src/mydialog.cpp.  
#include <QtGui/QtGui>  
#include "logindlg.h"  
int main(int argc, char* argv[])  
{  
    QApplication app(argc, argv);  
    QTextCodec::setCodecForTr(QTextCodec::codecForName("gb18030"));  
  
    QTranslator translator;  
    {  
        QStringList environment = QProcess::systemEnvironment();  
        QString str;  
        bool bFinded = false;
```

QTranslator 类提供了对文本输出的国际化 (internationalization) 的支持。创建的 QTranslator 对象 translator 包含了一组从源语言到目标语言的翻译 (translations), 这些翻译是通过 QTranslator 对象参照 Qt 翻译文件来实现的。关于国际化的详细内容, 请参阅本书第 20 章“国际化”。

静态函数 QProcess::systemEnvironment() 获取进程的环境变量, 并将其存放在 QStringList 对象 environment 中。QStringList 类是存放 Qt 字符串 QString 对象的列表类。

接下来, 定义一个存放环境变量“QTDIR”的 Qt 字符串 QString 对象 str; 定义查找结果的标识符 bFinded, 如果找到环境变量“QTDIR”其值为 true, 否则为 false, bFinded 初始化为 false (变量的初始化是一个良好的编程习惯)。

```
foreach(str, environment)  
{  
    if(str.startsWith("QTDIR="))  
    {  
        bFinded = true;  
        break;  
    }  
}
```

foreach() 函数体的功能是在获取的环境变量列表中查找以“QTDIR=”开头的字符串, 即查找环境变量“QTDIR”。如果找到环境变量“QTDIR”则将其保存在 QString 对象 str 中并设置标识符 bFinded 为 true, 然后退出循环; 否则继续进行查找, 直到遍历完整个 environment 环境变量列表。

foreach (variable, container) 是 Qt 定义的一个宏, 它的作用是遍历 QList 容器类变量 container 并依次把列表项存放在变量 variable 中。此外, Qt 还提供了功能等价的宏变量 Q_FOREACH (variable, container), 不同的是 foreach() 宏变量是可以通过配置 qmake 工程文件屏蔽掉的 (CONFIG += no_keywords), 而 Q_FOREACH() 却总是可以用。

```
if(bFinded)  
{
```

```

str = str.mid(5);
bFinded = translator.load("qt_" + QLocale::system().name(),
                          str.append("/translations/"));
if(bFinded)
    qApp->installTranslator(&translator);
else
    qDebug() << QObject::tr("没有支持中文的 Qt 国际化翻译文件!");
}
else
{
    qDebug() << QObject::tr("必须设置 QTDIR 环境变量!");
    exit(1);
}
}

```

这段代码的功能是给应用程序安装翻译器 (translator)。

if() 语句首先判断是否已经查找到环境变量 “QTDIR”。如果存在环境变量 “QTDIR”，则通过 QString::mid() 函数取得环境变量的值 (去掉字符串 “QTDIR=” 后剩下的内容)，即 Qt 4.3 的安装路径。

函数 QTranslator::load(const QString& filename, const QString& directory = QString(), const QString& search_delimiters = QString(), const QString& suffix = QString()) 加载 Qt 的翻译文件 (translation file)。

- 形参 filename+形参 suffix (如果没有指定 suffix 的实参，则 suffix 使用默认值 “.qm”) 指定加载的文件名称；
- 形参 directory 指定加载的路径；
- 形参 search_delimiters 指定翻译文件名字的过滤器 (如果 search_delimiters 为空，其默认值为 “_”)。

函数 QLocale::system().name() 以 “语言_国家” 的形式返回系统场景 (system locale，安装 Linux 操作系统的时候已经确定了系统的场景) 的名字 (“语言” 表示为两个小写字母的 ISO 639 语言代码，简体中文的语言代码为 zh，“国家” 表示为两个大写字母的 ISO 3166 国家代码，中国的国家代码为 CN)。因此，在登录对话框应用程序中，相当于调用了 QTranslator::load("qt_zh_CN", "\$QTDIR/translations/");

该函数将以下列规则加载 Qt 翻译文件 (在参数 directory 指定的目录下)：

- (1) 查找具有后缀 suffix (.qm) 文件名的文件，即查找 qt_zh_CN.qm；
- (2) 如果文件 qt_zh_CN.qm 不存在，查找没有后缀 suffix (.qm) 文件名的文件，即查找 qt_zh_CN；
- (3) 如果文件 qt_zh_CN 不存在，查找 search_delimiters (因为没有指定 search_delimiters 的实参，所以 search_delimiters 值为默认值 “_”) 定界符之前的字符加后缀 suffix 构成文件名的文件，即查找 qt_zh.qm；

- (4) 如果文件 qt_zh.qm 不存在，查找没有后缀 suffix (.qm) 文件名的文件，即查找 qt_zh。

- (5) 重复 (3) 和 (4)，即查找 qt.qm 和 qt。

在上述过程中，如果 QTranslator::load() 函数加载成功，返回 true；否则返回 false。事实上，应用程序的翻译器在步骤 1 就已经查找到了 Qt 翻译文件 qt_zh_CN.qm (文件位置在 Qt 安装路径的 translations 目录下。例如，Linux 系统下为 /usr/local/Trolltech/Qt-4.3.2/translations)，并成功地进行了加载。

函数 QApplication::installTranslator() 为应用程序添加翻译文件 (通过翻译器指定的翻译文件)。一个应用程序可以添加多个翻译文件，当应用程序需要进行翻译的时候，按照后添加先使用的原则依次在翻译文件中搜索需要的翻译文本，一旦找到一个匹配的翻译文件应用程序就停止进行搜索。



程序中，使用了大括号把加载/安装翻译文件的代码括了起来，这样做是为了及时销毁以后不再使用的栈对象，这在大型的应用程序开发中是有必要的。

```
CLoginDlg dlg;
return dlg.exec();
}
```

创建一个 CLoginDlg 对话框的栈对象 dlg。

槽函数 QDialog::exec() 以模态方式显示对话框对象 dlg，并进入对话框对象 dlg 的事件循环。当登录对话框运行完毕后将返回它的运行结果。注意，此处并没有启动 Qt 应用程序 QApplication 对象的事件循环，尽管也定义了一个 QApplication 对象 app（Qt GUI 应用程序必须有且仅有一个 QApplication 对象）。

对话框分为两种类型

- 模态（modal）对话框。
- 非模态（modeless）对话框。

模态对话框类型是最普遍的对话框类型。模态对话框在没有消失之前用户不能够与同一个应用程序的其他窗口进行交互，直到该对话框关闭。对于非模态对话框，当被打开时，用户既可以选择同该对话框进行交互，也可以选择同应用程序的其他窗口进行交互。

非模态对话框如果是栈对象，当代码退出对话框对象的作用域后，该对话框就自动销毁了，这就造成用户来不及和对话框进行交互，对话框就消失了。因此，必须通过 new 操作在堆中创建非模态对话框。

Qt 中，QDialog::exec() 以模态方式显示对话框，而 QDialog::show() 默认以非模态方式显示对话框。

最后，编译、连接登录对话框应用程序。程序的运行界面如图 2-13 和图 2-14 所示。

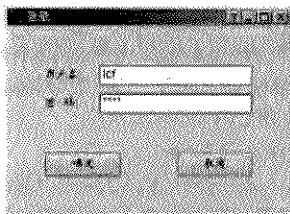


图 2-13 输入用户名和密码

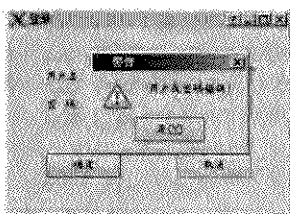


图 2-14 用户名和密码的验证

2.3 Qt 内建（built-in）对话框

为了便于程序员进行某些特定功能对话框的编程，Qt 提供了一套标准的通用对话框。这些内建的对话框都提供了一些便于使用的静态函数，在 Windows 系统上，这些静态函数将会调用本地的 Windows 对话框；而在 Mac OS X 系统下，这些静态函数将会调用本地的 Mac OS X 对话框。

除了上面已经详细介绍的 QMessageBox 消息框之外，Qt 提供的内建对话框还有：

- 颜色对话框 QColorDialog，能够允许用户选择颜色，效果如图 2-15 所示。
- 错误消息框 QErrorMessage，显示错误信息，效果如图 2-16 所示。

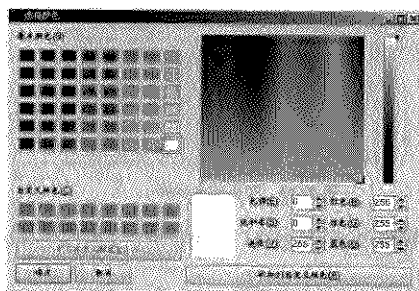


图 2-15 “选择颜色”对话框

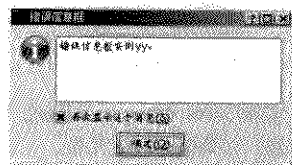


图 2-16 错误消息框

- 文件对话框 QFileDialog，能够允许用户选择一个或者多个文件以及目录，如图 2-17 所示。

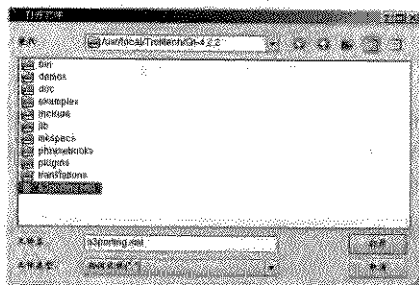


图 2-17 “打开文件”对话框

- 字体对话框 QFontDialog，允许用户选择/设置字体，效果如图 2-18 所示。

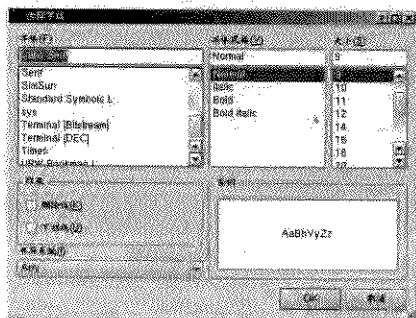


图 2-18 “选择字体”对话框

- 输入对话框 QInputDialog，允许用户进行简单的输入，比如输入一行文本或者一个数字等，效果如图 2-19 所示。
- 页设置对话框 QPageSetupDialog，配置与页相关的打印机选项，效果如图 2-20 所示。



-
- A number line from 0 to 10. The number 5 is circled, and an arrow points from 5 to 10.

level

特点

/ ໓໐ /

```

class QTextEdit;
class QPushButton;
class CBuiltInDlg : public QDialog
{
    Q_OBJECT
public:
    CBuiltInDlg(QWidget* = 0);
    virtual ~CBuiltInDlg() {}
private:
    QTextEdit* displayTextEdit;
    QPushButton* colorPushBtn;
    QPushButton* errorPushBtn;
    QPushButton* filePushBtn;
    QPushButton* fontPushBtn;
    QPushButton* inputPushBtn;
    QPushButton* pagePushBtn;
    QPushButton* progressPushBtn;
    QPushButton* printPushBtn;
private slots:
    void doPushBtn();
};
#endif

```

在类 `CBuiltInDlg` 的私有区，声明了一个文本编辑框，用来存放显示实例的文本信息；接下来，声明了控制 Qt 内建对话框显示的 `QPushButton` 按钮对象，共 7 个。

私有槽函数 `doPushBtn()` 响应 `QPushButton` 按钮对象的单击操作信号 `clicked()`。

接下来，看一下自定义的内建对话框类的实现文件。

首先将必要的头文件包含进来。

```

// chapter02/builtin/src/builtindlg.cpp
#include <QtGui/QtGui>
#include "builtindlg.h"

```

下面是内建对话框类 `CBuiltInDlg` 的构造函数。

```

CBuiltInDlg::CBuiltInDlg(QWidget* parent)
: QDialog(parent)
{
    displayTextEdit = new QTextEdit(tr("Qt 的标准通用对话框。"));

```

创建一个显示内容为“Qt 的标准通用对话框。”的文本编辑框 `QTextEdit` 对象，该文本编辑框将在演示 Qt 的各种内建对话框时用到。

```

QGridLayout* gridLayout = new QGridLayout;
colorPushBtn = new QPushButton(tr("颜色对话框"));
errorPushBtn = new QPushButton(tr("错误消息框"));
filePushBtn = new QPushButton(tr("文件对话框"));
fontPushBtn = new QPushButton(tr("字体对话框"));
inputPushBtn = new QPushButton(tr("输入对话框"));

```



```
pagePushBtn    = new QPushButton(tr("页设置对话框"));
progressPushBtn = new QPushButton(tr("进度对话框"));
printPushBtn    = new QPushButton(tr("打印对话框"));
gridLayout->addWidget(colorPushBtn, 0, 0, 1, 1);
gridLayout->addWidget(errorPushBtn, 0, 1, 1, 1);
gridLayout->addWidget(filePushBtn, 0, 2, 1, 1);
gridLayout->addWidget(fontPushBtn, 1, 0, 2, 1);
gridLayout->addWidget(inputPushBtn, 1, 1, 1, 1);
gridLayout->addWidget(pagePushBtn, 1, 2, 1, 1);
gridLayout->addWidget(progressPushBtn, 2, 0, 1, 1);
gridLayout->addWidget(printPushBtn, 2, 1, 1, 1);
gridLayout->addWidget(displayTextEdit, 3, 0, 3, 3);

setLayout(gridLayout);
```

创建一个网格布局管理器 `QGridLayout` 对象, `gridLayout` 布局管理器将会管理和排布所有的窗口部件。

接下来, 创建 7 个 `QPushButton` 对象, 这些对象分别用来控制颜色对话框、错误消息框、文件对话框、字体对话框、输入对话框、页设置对话框、进度对话框和打印对话框的创建和显示。

然后调用 `QGridLayout::addWidget()` 函数, 将所有的 `QPushButton` 以及 `QTextEdit` 窗口部件排布在网格布局管理器 `gridLayout` 中。

最后, 函数 `QDialog::setLayout()` 将网格布局管理器 `gridLayout` 设置为内建对话框 `CBuiltinDlg` 对象的顶层布局管理器。

```
connect(colorPushBtn, SIGNAL(clicked()), this, SLOT(doPushBtn()));
connect(errorPushBtn, SIGNAL(clicked()), this, SLOT(doPushBtn()));
connect(filePushBtn, SIGNAL(clicked()), this, SLOT(doPushBtn()));
connect(fontPushBtn, SIGNAL(clicked()), this, SLOT(doPushBtn()));
connect(inputPushBtn, SIGNAL(clicked()), this, SLOT(doPushBtn()));
connect(pagePushBtn, SIGNAL(clicked()), this, SLOT(doPushBtn()));
connect(progressPushBtn, SIGNAL(clicked()), this, SLOT(doPushBtn()));
connect(printPushBtn, SIGNAL(clicked()), this, SLOT(doPushBtn()));
```

将所有的 `QPushButton` 对象的 `clicked()` 信号关联到内建对话框类 `CBuiltinDlg` 的槽函数 `doPushBtn()`, 即所有 `QPushButton` 对象的单击操作都由统一的槽函数 `doPushBtn()` 来处理。

```
setWindowTitle(tr("内建对话框"));
resize(400, 300);
}
```

函数 `QDialog::setWindowTitle()` 设置对话框的标题为“内建对话框”。

最后, 调用函数 `QDialog::resize()` 改变对话框的大小尺寸。

下面, 看一下槽函数 `doPushBtn()`, 它接收并处理所有的 `QPushButton` 按钮的单击信号 `clicked()`。

```
void CBuiltinDlg::doPushBtn()
{
    QPushButton* btn = qobject_cast<QPushButton*>(sender());
    if(btn == colorPushBtn)
    {
        // 颜色对话框。
        QPalette palette = displayTextEdit->palette();
```

```

const QColor& color =
    QColorDialog::getColor(palette.color(QPalette::Base), this);
if(color.isValid())
{
    palette.setColor(QPalette::Base, color);
    displayTextEdit->setPalette(palette);
}
}

```

槽函数的开头，首先是获取发送信号的 QPushButton 对象的指针。函数 QObject::sender() 返回发送信号的对象指针，返回类型为 QObject*。模板函数

`T qobject_cast(QObject* object)`

完成类型的转换，将 `<QObject*>` 类型的对象指针转换为类型为 `<T*>` 的对象指针，如果转换成功，返回正确的对象指针，否则返回 0。注意，类型 T 必须是直接或间接继承自 QObject 的类，并且在该类的定义里有 Q_OBJECT 宏变量（否则 qobject_cast() 函数的返回值是未定义的）。此外，可以认为一个类继承自它自身。qobject_cast() 模板函数的作用类似于标准 C++ 的 dynamic_cast() 模板函数，不过 qobject_cast() 不需要运行时类型信息（Run-Time Type Information, RTTI）的支持。

接下来是 if() 条件判断语句，判断发送信号的对象是否是相应的 QPushButton 对象（colorBtn、errorPushBtn 等），如果是则创建相应的 Qt 内建对话框并进行显示；否则将会跳过该段代码，直到找到条件为 true 的 if() 条件语句。

上面这段代码是颜色对话框的例子，它的功能是利用 Qt 内建的颜色对话框 QColorDialog 类获取用户选择的颜色，然后设置文本编辑框的背景色。

函数 QTextEdit::palette() 获取文本编辑框对象 displayTextEdit 的调色板。

接下来，调用 QColorDialog::getColor() 函数创建并显示一个模态的颜色对话框，并返回用户选择的颜色对象的常量引用（对临时对象的引用是无效的，必须使用常量引用）赋给变量 color；如果用户单击“取消”按钮，颜色对话框将返回一个无效的颜色；颜色对话框的颜色初始化为 palette.color(QPalette::Base)，即文本编辑框的背景色；父窗口部件为 this。

QColor::isValid() 函数判断颜色对话框返回的颜色是否有效。如果颜色对话框返回的颜色是有效的，函数 QPalette::setColor() 设置调色板的背景颜色为颜色对话框返回的颜色。此处的 QPalette::setColor() 函数具有 2 个参数，第 1 个参数 QPalette::Base 指定了调色板的角色，告诉程序应该设置调色板的所有三个颜色组中的哪一个角色的颜色（该函数将会影响到所有的三个颜色组）；第 2 个参数 color 指定应该设置的颜色。最后，函数 QTextEdit::setPalette() 重新设置文本编辑框的调色板。



延伸阅读

关于 Qt 调色板类 QPalette

QPalette 类包含了 Qt 窗口部件的颜色组（color group）：

- Active 组，该组的颜色用于当前活动的（active）窗口，即具有键盘或鼠标焦点的窗口；
- Inactive 组，该组用于其他的窗口；
- Disabled 组，该组用于状态为不可用的（disabled）的子窗口部件（不包含窗口）。

所有 Qt 窗口部件都拥有一个调色板并使用它绘制自己。通常，活动状态的窗口的标题栏显示为蓝



色的,而非活动(inactive)状态的窗口的标题栏显示为灰色的;活动状态的窗口和非活动状态的窗口都可以包含状态为不可用的窗口部件,一个不可用的窗口部件(包括该窗口部件包含的子窗口部件)显示为灰色的,用户是无法同它进行交互的。通过改变窗口部件的调色板的各个组中的颜色,能够改变窗口部件的显示颜色,比如改变背景色、文本颜色等。

可以通过 `QWidget::palette()` 获取一个窗口部件的调色板,然后通过 `QWidget::setPalette()` 函数为该窗口部件设置修改后的调色板。或者通过 `QApplication::palette()` 函数获取应用程序的调色板,并通过 `QApplication::setPalette()` 为该应用程序设置修改后的调色板。修改一个窗口部件的调色板只会影响到该窗口部件以及子窗口部件(不包含子窗口);而改变一个应用程序的调色板将会影响到该应用程序的所有窗口部件。当对一个窗口部件的调色板已经作了修改后,再对其父窗口部件调色板的修改不会影响到该窗口部件的调色板;同样,对应用程序调色板的修改不会影响已经单独作出了调色板修改的窗口部件。

调色板类 `QPalette` 提供了颜色角色(color roles)概念,是指当前 GUI 界面中颜色的职责,通过枚举变量 `QPalette::ColorRole` 来定义,比较常用的颜色角色有:

- `QPalette::Window`, 通常指窗口部件的背景色;
- `QPalette::WindowText`, 通常指窗口部件的前景色;
- `QPalette::Base`, 指文本输入窗口部件(比如 `QTextEdit`、`QLineEdit` 等)的背景色,上面的例子中使用了该颜色角色;
- `QPalette::Text`, 与 `QPalette::Base` 一块使用,指文本输入窗口部件的前景色;
- `QPalette::Button`, 指按钮窗口部件的背景色;
- `QPalette::ButtonText`, 指按钮窗口部件的前景色。

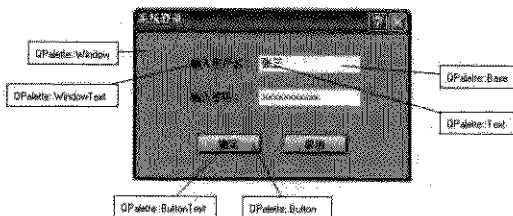


图 2-23 颜色角色

```
else if(btn == errorPushBtn)
{
    // 错误消息框。
    QMessageBox box(this);
    box.setWindowTitle(tr("错误消息框"));
    box.showMessage(tr("错误消息框实例 xx. "));
    box.showMessage(tr("错误消息框实例 xx. "));
    box.showMessage(tr("错误消息框实例 xx. "));
    box.showMessage(tr("错误消息框实例 yy. "));
    box.showMessage(tr("错误消息框实例 zz. "));
    box.exec();
}
```

该段代码是一个错误消息框的例子。在这个例子中,多次调用了 `QErrorMessage::showMessage()`

函数，该函数的功能是在错误消息框中显示错误信息。多次调用该函数，是为了演示显示不同错误信息、多个相同错误信息以及错误消息框的“再次显示这个消息”复选框选中与否的效果。最后，执行 `QErrorMessage::exec()` 显示对话框。此外，还可以 `new` 一个错误消息框对象，仅仅通过 `QErrorMessage::showMessage()` 函数就可以显示对话框，而无需执行 `QErrorMessage::exec()` 函数。

```
else if(btn == filePushBtn)
{
    // 文件对话框。
    QString fileName = QFileDialog::getOpenFileName(
        this,
        tr("打开文件"),
        "/usr/local/Trolltech",
        tr("任何文件(*.*)"),
        tr(";;文本文件(*.txt)");
        tr(";;XML文件(*.xml)"));
    displayTextEdit->setText(fileName);
}
```

该段代码打开一个文件对话框，获取用户选择的文件名并显示在文本编辑框中。此处，函数 `QFileDialog::getOpenFileName()` 具有 4 个参数。其中，实参 `this` 指定文件对话框的父窗口部件；实参 `tr("打开文件")` 指定文件对话框的标题；实参 `"/usr/local/Trolltech"` 指定了文件对话框的默认路径；最后一个参数比较复杂，它指定了文件对话框的多个文件过滤器，过滤器之间通过两个分号“;;”间隔。如果用户选择了文件，并单击“确定”按钮，那么该文件对话框将返回用户选择的文件名；而如果用户单击“取消”按钮，该对话框将返回一个 `NULL` 字符串。

```
else if(btn == fontPushBtn)
{
    // 字体对话框。
    bool ok;
    const QFont& font = QFontDialog::getFont(&ok,
        displayTextEdit->font(),
        this,
        tr("字体对话框"));
    if(ok)
    {
        // 如果<确定>, 设置字体。
        displayTextEdit->setFont(font);
    }
}
```

`QFontDialog::getFont()` 函数创建并显示一个字体对话框。此处，该函数具有四个参数。第 1 个参数是一个输出参数，用于标识用户的选择状态，如果用户单击“确定”按钮，该字体对话框将会设置 `ok` 变量为 `true`；而如果用户单击“取消”按钮，`ok` 将会被设置为 `false`。第 2 个参数指定了对话框的初始颜色，当用户取消颜色的选择时，字体对话框将初始颜色作为返回值。`this` 参数指定了父窗口部件，最后一个参数指定了字体对话框的标题。

```
else if(btn == inputPushBtn)
{
    // 输入对话框。
    bool ok;
    QString text = QInputDialog::getText(this,
        tr("输入对话框"),
```

```

        tr("输入文本: ") ,
        QLineEdit::Normal,
        QDir::home().dirName(),
        &ok);
    if (ok && !text.isEmpty())
        displayTextEdit->setText(text);
}

```

`QInputDialog::getText()` 函数创建并显示一个文本输入对话框。我们赋给该函数 6 个参数，前 2 个参数分别指定了输入对话框的父窗口部件和对话框的标题；第 3 个参数指定了标签的显示文本；第 4 个参数指定了行编辑器 `QLineEdit` 输入内容的显示模式，此处为 `QLineEdit::Normal`，即按用户输入的实际内容显示；第 5 个参数指定了行编辑框默认显示的内容，函数 `QDir::home()` 返回用户的 home 路径，`QDir::dirName()` 返回路径的名字；最后一个参数和 `QFontDialog::getFont()` 函数的第 1 个参数的作用相同。

```

else if (btn == pagePushBtn)
{
    // 页设置对话框。
    QPrinter printer;
    QPageSetupDialog dlg(&printer, this);
    dlg.setWindowTitle(tr("页设置对话框"));
    if (dlg.exec() == QDialog::Accepted)
    {
        // 进行下一步的处理。
    }
}
}

```

首先，定义了一个打印机 `QPrinter` 对象 `printer`。然后创建了一个页设置对话框 `QPageSetupDialog` 对象，并设置对话框的标题。在这个例子中，只是简单地创建和显示一个页设置对话框，该对话框返回后没有进行下一步的处理。

```

else if (btn == progressPushBtn)
{
    // 进度对话框。
    QProgressDialog progress(tr("正在复制文件..."),
                             tr("取消"),
                             0,
                             10000,
                             this);
    progress.setWindowModality(Qt::WindowModal);
    progress.setWindowTitle(tr("进度对话框"));
    progress.show();
    for (int i = 0; i < 10000; i++)
    {
        progress.setValue(i);
        QApplication->processEvents();
        if (progress.wasCanceled())
            break;
        //... 复制文件处理。
    }
}

```



```

        qDebug() << i;
    }
    progress.setValue(10000);
}

```

这一段代码创建了一个进度对话框，并模拟了显示一个长时间操作的工作进程。

首先，调用了 `QProgressDialog` 的构造函数，创建了一个进度对话框的栈对象。该构造函数有 5 个参数。第 1 个参数是一个字符串，它指定了显示给用户的提示信息，表明当前正在进行的工作。第 2 个参数指定了“取消”按钮的显示文本。如果该参数为 0 的话，进度对话框将没有“取消”按钮，即创建进度对话框的代码为

```
QProgressDialog progress(tr("正在复制文件..."), 0, 0, 10000, this);
```

接下来的 2 个参数分别指定了操作的步数（在上面的例子中，可以假定进度对话框显示复制 10000 个文件的进展情况，第 3 个参数设定为 0，第 4 个参数设定为 10000，即这两个参数的差值决定了这个复制操作的步数为 10000）。第 5 个参数指定了进度对话框的父窗口部件。

接下来，函数 `setWindowModality()` 设置进度对话框的类型为 `Qt::WindowModal`，即为模态对话框。有两种方式使用进度对话框 `QProgressDialog`：

- **模态对话框方式。**使用一个模态进度对话框显示长时间操作的工作进度对于编程是比较简单的，但是必须调用 `QApplication::processEvents()` 函数或者 `QEventLoop::processEvents()` (`QEventLoop::ExcludeUserInputEvents`) 函数以保证事件循环还可以处理其他事件，以防止应用程序因为长时间的操作而导致没有反应。在自定义对话框的例子中，使用了模态进度对话框，通过 `QProgressDialog::setValue()` 函数推进显示的进度；通过 `QProgressDialog::wasCanceled()` 函数判断用户是否中途选择了“取消”按钮，如果是，将中断复制文件操作。此外，代码使用了 `qDebug()` 函数打印 `for()` 语句的运行进度，模拟复制操作。
- **非模态对话框方式。**非模态进度对话框适用于显示在后台运行的长时间操作的工作进度，这样的话，用户就能够和应用程序的其他窗口进行交互。

```

else if(btn == printPushBtn)
{
    // 打印对话框。
    QPrinter printer;
    QPrintDialog dlg(&printer, this);
    dlg.setWindowTitle(tr("打印对话框"));
    if (dlg.exec() == QDialog::Accepted)
    {
        // 进行下一步的处理。
    }
}
}

```

创建并显示打印对话框的例子很简单，在对话框返回后没有进行下一步的处理。

现在修改主程序文件 `builtin.cpp`（内容基本和 `mydialog.cpp` 相同，不同的是把包含的头文件替换为“`builtinDlg.h`”，对话框类改为“`CBuiltinDlg`”），并编译运行内建对话框应用程序，显示效果如图 2-23 所示。

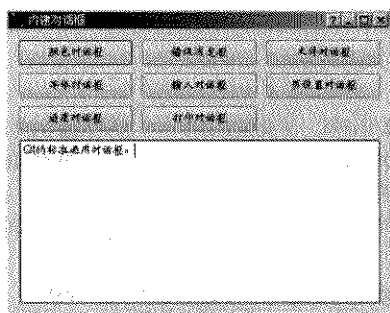


图 2-24 内建对话框

2.4 小 结

这一章,学习了 Qt 对话框的基本概念,学习了如何继承 `QDialog` 对话框类实现一个登录对话框类;介绍了如何利用 Qt 提供的国际化翻译文件将 Qt 的内建对话框文本内容显示为中文字符;此外,还概述了 Qt 的几种 Qt 内建对话框的含义,举例介绍了它们的使用。

第3章 基础窗口部件——QWidget

在前面的章节中，学习了直接继承 Qt 对话框类 `QDialog` 实现自定义的对话框，并通过手写代码实现了子窗口部件的添加和排布。从本章开始，将学习使用 Qt 的 GUI 界面设计器（Qt Designer）进行界面的绘制和布局，学习如何在应用程序中加载和使用 Qt 设计器绘制的窗口部件；学习 Qt 的基础窗口部件——`QWidget` 的基本特性和功能以及 Qt 样式表，并深入地介绍 Qt 元对象系统、属性系统和对象树机制。

3.1 Qt 设计器绘制窗口部件

Qt 设计器是 Qt GUI 编程语言一系列工具中的一个，该工具提供了 Qt 基本的可绘制窗口部件，比如 `QWidget`、`QLabel`、`QPushButton`、`QVBoxLayout` 等。在设计器中通过鼠标直接拖放这些窗口部件，能够高效、快速地完成 GUI 界面的设计，界面直观形象，所见所得。

本节将从绘制一个自定义的 `QWidget` 窗口部件来学习 Qt 设计器的使用，同时也将了解 Qt 的资源文件（*.qrc）、ui 文件（*.ui）以及相应的编译工具——资源编译器（Resource Compiler, `rcc`）和用户界面编译器（User Interface Compiler, `uic`）。

3.1.1 Qt 设计器基础

首先，将 Qt 设计器打开。在控制台执行命令

```
designer&
```

其中，“&”表示将在后台运行 Qt 设计器，这样就可以继续使用该控制台运行其他的命令。进入 Qt 设计器的默认启动界面，如图 3-1 所示（最新的 Qt 4.3.2 带的 Qt 设计器是中文界面）。

在“New Form”对话框中，可以选择要设计的顶层窗口部件类型（顶层窗口部件是其他子窗口部件的载体）：

- Dialog with Buttons Bottom，底部具有“确定”和“取消”按钮的对话框；
- Dialog with Buttons Right，右侧具有“确定”和“取消”按钮的对话框；
- Main Window，应用程序主窗口界面；
- Widget，Qt 的基础窗口部件 `QWidget`。

“New Form”对话框的右侧是窗口部件类型（template / forms）的预览区。

进入 Qt 设计器主界面后，就可以使用 Qt 的标准窗口部件进行 GUI 可视化设计了。在 Qt 设计器的左侧“Widget Box”栏列出了经常使用的 Qt 标准窗口部件，单击鼠标左键选中相应的窗口部件图标，可以将它们直接拖放到顶层窗口部件的界面上。同时，也可以将自己设计的窗口部件组合（通过使用布局管理器对 Qt 标准窗口部件进行布局和组合）或放置了其他窗口部件的 Qt 容器类（见“Widget Box”栏的“Containers”组）直接拖放到“Widget Box”栏中，Qt 设计器会自动在“Widget Box”栏中生成



“Scratchpad”组，并生成新的自定义的窗口部件。以后可以像使用 Qt 提供的标准窗口部件一样使用自己创建的窗口部件。

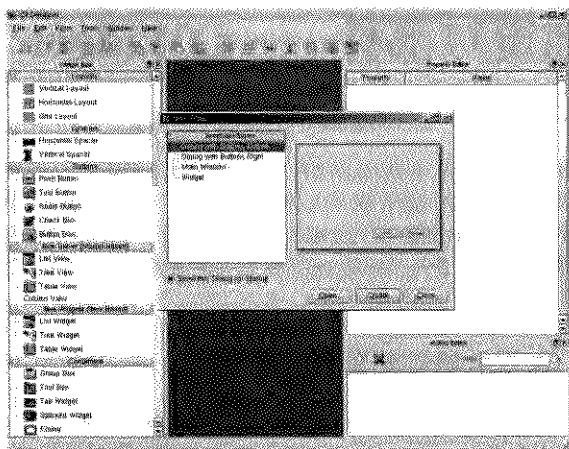


图 3-1 Qt 设计器初始界面

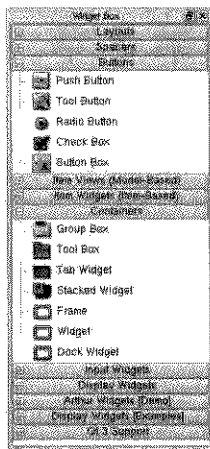
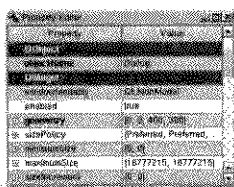
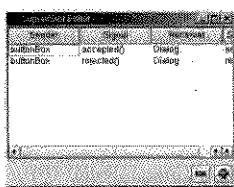


图 3-2 Qt 设计器中的部件

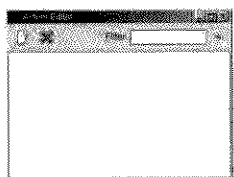
选中 Qt 设计器 Tools 菜单中的全部复选框，在 Qt 设计器的右侧可以看到设计器提供的一些编辑工具，如图 3-3 所示。



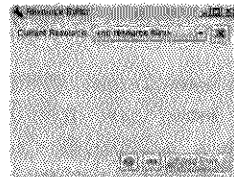
属性编辑器



信号/槽编辑器



动作编辑器



资源编辑器



对象监视器

图 3-3 Qt 设计器提供的编辑器

- **对象监视器 (Object Inspector)**: 列出了界面中的所有窗口部件, 以及各窗口部件的父子关系和包含关系。
- **属性编辑器 (Property Editor)**: 列出了窗口部件可编辑的属性。
- **动作编辑器 (Action Editor)**: 列出了为窗口部件设计的 QAction 动作, 通过“添加”或“删除”按钮可以新建一个可命名的 QAction 动作或删除指定的 QAction 动作。
- **信号/槽编辑器 (Signal/Slot Editor)**: 列出了在 Qt 设计器中关联的信号和槽, 通过双击列中的对象或信号/槽, 可以进行对象的选择和信号/槽的选择。
- **资源编辑器 (Resource Editor)**: 列出了程序使用的资源文件 (*.qrc) 以及相应的资源, 在该编辑器中可以创建或删除资源文件, 也可以添加或取消资源。

此外, 通过 Qt 设计器的“Edit”菜单, 可以打开 Qt 设计器的 4 种 GUI 窗口部件编辑模式:

- **Widget 编辑模式 (Widget Editing Mode)**: 可以在 Qt 设计器中添加 GUI 窗口部件以及修改它们的属性和外观。
- **信号和槽编辑模式 (Signals and Slots Editing Mode)**: 可以在 Qt 设计器中的窗口部件上关联 Qt 已经定义好的信号和槽。
- **伙伴编辑模式 (Buddy Editing Mode)**: 可以在 Qt 设计器中的窗口部件上建立 QLabel 标签和其他窗口部件的伙伴关系。
- **Tab 编辑模式 (Tab Order Editing Mode)**: 可以在 Qt 设计器中的窗口部件上设置 Tab 键在窗口部件上的焦点顺序。

扩展阅读

QLabel 标签和伙伴 (buddy) 窗口部件

一个标签 QLabel 和一个窗口部件具有伙伴关系, 即指当用户激活标签的快捷键时, 鼠标/键盘的焦点将会转移到它的伙伴窗口部件上。只有 QLabel 标签对象才可以有伙伴窗口部件, 也只有该 QLabel 对象具有快捷键 (在显示文本的某个字符前面添加一个前缀“&”, 就可以定义快捷键) 时, 伙伴关系才有效。例如,

```
QLineEdit* priceLineEdit = new QLineEdit(this);
QLabel* priceLabel = new QLabel("&Price", this);
priceLabel->setBuddy(priceLineEdit);
```

代码定义了 priceLabel 标签的快捷键为“Alt + P”, 并将行编辑框 priceLineEdit 设为它的伙伴窗口部件。所以当用户按键“Alt+P”时, 焦点将会跳至行编辑框 priceLineEdit 中。

Qt 设计器提供了伙伴编辑模式, 可以通过鼠标拖放操作快捷地建立标签 QLabel 和其他窗口部件的伙伴关系。

对于 Qt 设计器提供的上述工具和编辑模式, 将在下一节学习。在此, 只对 Qt 资源系统 (Qt Resource System) 略加介绍。

Qt 资源系统是平台无关的, 它以二进制代码保存可执行应用程序运行时使用的文件 (如图标文件)。在编译 Qt 应用程序时, Qt 资源编译器 rcc 会对 Qt 资源文件进行编译并生成相应的二进制代码文件 (最后由连接程序将该二进制代码文件连接为应用程序的一部分), 而该资源文件一旦通过编译就不能够再改变, 除非重新编译 Qt 应用程序。因此, 需要运行时改变的文件是不能够放置在 Qt 的资源系



统中的（例如，在程序运行期间需要编辑修改的 XML 文件）。

Qt 资源文件*.qrc 是 XML 格式的文本文件，它记录了 Qt 应用程序使用的资源。资源文件及其记录的资源是应用程序源码的一部分，一个应用程序可以有多个资源文件。注意，资源文件*.qrc 中记录资源的路径是资源相对于资源文件*.qrc 的位置。

对于上述目录结构描述的资源，其资源文件 hello.qrc 的内容为：

```
<!DOCTYPE RCC>
<RCC version="1.0">
  <qresource>
    <file>lines/dotline.png</file>
    <file>lines/dashdotline.png</file>
    <file>lines/solidline.png</file>
    <file>lines/dashline.png</file>
  </qresource>
</RCC>
```

这是一个 XML 格式的文本文件。

第 1 行 XML 语句声明了 XML 文档的类型为 RCC，即 Qt 资源。

<RCC version="1.0"> 引入了 XML 文档的第一个元素 RCC，并声明了 RCC 的版本为 1.0。

元素<qresource>指示其子元素是资源元素。四行<file>元素记录了应用程序使用的资源，这些资源放置在 lines 子目录下（如图 3-4 所示）。

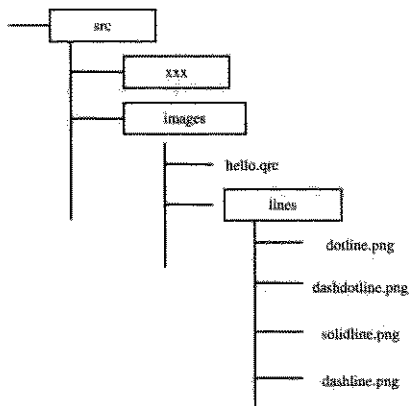


图 3-4 资源文件及其资源

使用资源系统之前，必须将定义的资源文件 *.qrc 引入到 qmake 工程文件中。对于 hello.qrc 则必须在 qmake 工程文件中加入下列一行：

```
RESOURCES += hello.qrc
```

它表示应用程序用到了资源文件 hello.qrc，并告知 qmake 工具。

接下来，就可以在应用程序启动的时候通过使用 Q_INIT_RESOURCE() 函数进行资源的初始化。

```
int main(int argc, char *argv[])
```

```

{
    QApplication app(argc, argv);
    Q_INIT_RESOURCE(hello);
    ...
    return app.exec();
}

```

宏 `Q_INIT_RESOURCE()` 初始化 `hello.qrc` 记录的资源，“hello”是资源文件的名称。之后，就可以在应用程序中引用资源了（应用程序的资源将在应用程序启动的时候自动加载）。例如，创建一个带有实线图标的动作对象：

```
QAction* act = new QAction(QIcon(":/lines/solidline.png"), tr("实线"), this);
```

在上述创建 `QAction` 动作对象的时候，使用了 `QAction(const QIcon& icon, const QString& text, QObject* parent)` 构造函数，它的形参 `icon` 和 `text` 都是对常量对象的引用，因此可以传入不具名的 `QIcon(":/lines/solidline.png")` 对象作为实参和临时 `QString` 对象 `tr("实线")` 作为实参。而对于非常量对象的引用，不具名的对象、临时对象和具体数值是不能够作为实参的。例如，如果其构造函数定义为 `QAction(QIcon& icon, QString& text, QObject* parent)`，那么上述 `QAction` 对象的一个合法的构造方法是：

```

QIcon icon(":/lines/solidline.png");
QString str(tr("实线"));
QAction* act = new QAction(icon, str, this);

```

对于 C++ 内嵌的数值性数据类型亦是如此。不过，很少有程序员将一个数值作为引用来传递，毕竟内嵌的简单数据类型的传值参数的开销是非常小的。

在定义好应用程序的资源/资源文件以及在程序中添加了初始化和加载资源的代码后，运行 `qmake` 生成 `Makefile` 文件，接着运行 `make` 命令编译/连接应用程序，`gcc` 编译器自动调用 Qt 的 `rcc` 资源编译器编译 `hello.qrc` 及其资源并生成 C++ 源文件 `qrc_hello.cpp`，最后 `gcc` 编译 `qrc_hello.cpp` 文件并同其他的目标代码连接成可执行的应用程序，如图 3-5 所示。

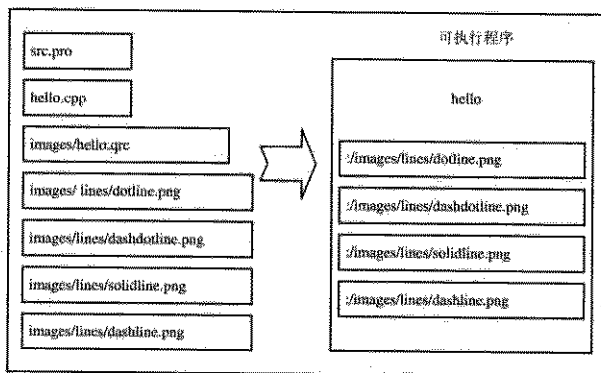


图 3-5 Qt 资源的编译/链接

此外，应用程序也可以使用外部的二进制资源（External Binary Resources），外部二进制资源不作



为应用程序的二进制代码的一部分。这种方式脱离于应用程序，可以在需要的时候进行动态加载。

要使用外部二进制资源，首先根据资源文件将资源编译成二进制的资源数据（文件的扩展名通常为 .rcc）。例如，上述的 hello.qrc 资源文件及其资源，将编译为二进制文件 hello.rcc。在控制台执行下列命令：

```
rcc -binary hello.qrc -o hello.rcc
```

然后，在应用程序中注册资源（假设二进制资源文件位于 /home/lcf/book/rcc/ 目录下）：

```
QResource::registerResource("/home/lcf/mybooks/rcc/hello.rcc");
```

接着就可以在应用程序中使用了，使用的方法同前面讲过的使用方法相同。

如果资源文件 .qrc 中记录的资源不存在，在编译的时候则会报错。而如果在应用程序中引用有错误时，编译不会报错，但在运行应用程序时会导致一些问题，例如无法显示图标等。

3.1.2 绘制窗口部件

上一节，讲到了 Qt 设计器的一些基础知识。现在，使用 Qt 设计器绘制一个实现查找文件功能的 GUI 窗口部件。

进入 Qt 设计器界面后，在“New Form”对话框的“template/forms”选项卡中选择“Widget”，单击“create”按钮。Qt 设计器将生成一个顶层的窗口部件，并以网格的方式显示该窗口部件。

将窗口部件的对象名字（objectName）属性修改为“FindFileForm”（Qt 的 uic 编辑器将根据顶层窗口部件的对象名字生成该界面的窗口部件类，将在下面讲到）。将窗口拉伸到一个合适的大小，或者通过设置窗口部件的“geometry”属性值改变窗口的大小。接下来，添加必要的子窗口部件，如图 3-6 所示。

按如下步骤添加各窗口部件：

01 添加“名为”、“包含文本”、“查找位置” QLabel

标签以及相关的编辑框（由上到下依次为 QComboBox、QLineEdit 和 QComboBox），添加一个“浏览...” QPushButton 按钮；设置它们的属性（如表 3-1 所示，通过属性编辑器设置），调整大小和位置，然后选中所有这些窗口部件（按 shift 键的同时用鼠标左键点选窗口部件），在选中的窗口部件上单击鼠标右键，选择上下文菜单“layout”的菜单命令“Layout in a Grid”，把它们放置在一个网格布局管理器中（或直接单击布局管理器工具栏中的“网格布局管理器”工具按钮，如图 3-7 所示）。

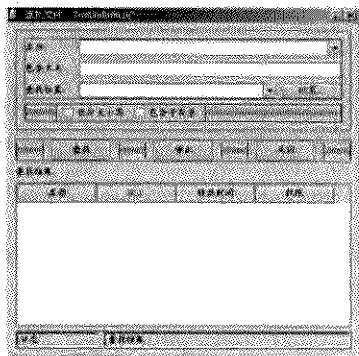


图 3-6 查找文件界面

表 3-1 窗口部件属性表（按从左到右、从上至下顺序，不包括布局管理器）

objectName	text	sizePolicy	editable	窗口部件类
label	名为：	hSizeType:fixed vSizeType:preferred	——	QLabel
nameComboBox	——	(默认)	true	QComboBox
label_2	包含文本：	hSizeType:fixed vSizeType:preferred	——	QLabel

续表

objectName	text	sizePolicy	editable	窗口部件类
txtLineEdit	——	(默认)	——	QLineEdit
label_3	查找位置:	hSizeType:fixed vSizeType:preferred	——	QLabel
dirComboBox	——	(默认)	true	QComboBox
browsePushBtn	——	hSizeType:Maximum vSizeType:fixed	——	QPushButton

图 3-7 布局管理器工具栏

图 3-7 布局管理器工具栏

排布后的各窗口部件如图 3-8 所示。

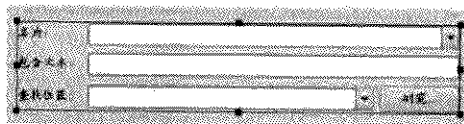


图 3-8 使用网格布局管理器组织输入窗口部件

02 添加两个 QCheckBox 复选框窗口部件：“区分大小写”复选框 sensitiveCheckBox（对象名字）和“包含子文件夹”复选框 subfolderCheckBox，并放置在一个水平布局管理器 QHBoxLayout 中，如图 3-9 所示。



图 3-9 使用水平布局管理器组织复选框窗口部件

为了界面的美观，使用了一些 Spacer 窗口部件（蓝色的像弹簧状的），选中这些 Spacer 可以在属性编辑器中设置它的 sizeType 属性。在 Qt 设计器中将 Spacer 加入到一个布局管理器，有两种办法：

- 选中各子窗口部件的同时也选中 Spacer，把它们件加入到布局管理器中；
- 将各子窗口部件加入一个布局管理器后，将 Spacer 拖放到该布局管理器的适当位置。

设置 Spacer 窗口部件的属性：

- 左侧 Spacer 的 sizeType 属性为 QSizePolicy::Fixed（其他采用默认值）；
- 右侧 Spacer 的属性全部采用默认值。

03 添加一个容器 QFrame 子窗口部件（属性采用默认值），并将步骤 **01**、**02** 生成的窗口部件组合依次拖放到该 QFrame 子窗口部件中，并设置它的布局管理器为垂直布局管理器，效果如图 3-10 所示。

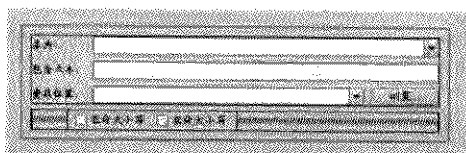


图 3-10 使用 QFrame 容器重新组织窗口部件

04 添加三个 QPushButton 窗口部件：“查找”按钮 findPushBtn、“停止”按钮 stopPushBtn 和“关闭”按钮 closePushBtn，并放置在一个水平布局管理器中；然后放入 4 个 Spacer 窗口部件，属性采用默认值，如图 3-11 所示。

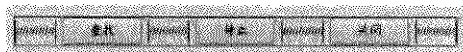


图 3-11 使用水平布局管理器组织按钮窗口部件

05 添加“查找结果”QLabel 标签和 QTableWidgetItem 表窗口部件 resultTableWidget，在表窗口部件中单击鼠标右键，在弹出的上下文菜单中选择“Edit Items...”命令，在出现的“Edit Table Widget”对话框中添加表列：名称、大小、修改时间和权限，如图 3-12 所示。

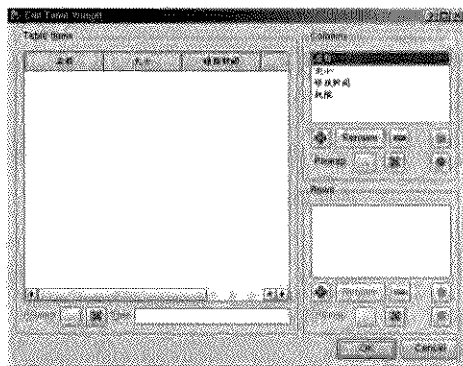


图 3-12 编辑表格窗口部件

06 添加“状态”标签 statusLabel 和“查找结果”标签 resultLabel，设置它们的 sizePolicy 属性（如表 3-2 所示），并把它们放置在一个水平布局管理器中。

表 3-2 窗口部件属性表

objectName	text	sizePolicy	窗口部件类
statusLabel	状态:	hSizePolicy:preferred vSizePolicy:preferred horizontalStretch:1 verticalStretch:0	QLabel
resultLabel	查找结果:	hSizePolicy:preferred vSizePolicy:preferred horizontalStretch:3 verticalStretch:0	QLabel

排布好的“状态”和“查找结果”标签如图 3-13 所示。

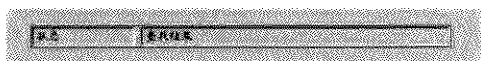


图 3-13 使用水平布局管理器组织状态显示标签

07 最后，选中顶层窗口部件（即最初建立的 QWidget 窗口部件），然后将整个窗口部件放置在一个垂直布局管理器中（如图 3-6 所示）。

现在使用 Qt 资源文件为应用程序用户界面的标题栏加一个标题和图标。

01 选择“Tools”菜单的“Resource Editor”命令，打开资源编辑器。

02 在“Current Resource”下拉框中选择“New...”选项（如图 3-14 所示），打开“新建资源文件”对话框。

03 在“新建资源文件”对话框中，定位到资源文件放置的目录（本例为 chapter03/findfile/），输入资源文件的名字 findfile.qrc，保存之，如图 3-15 所示。

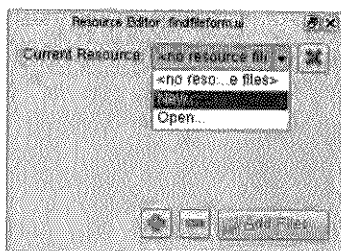


图 3-14 资源编辑器

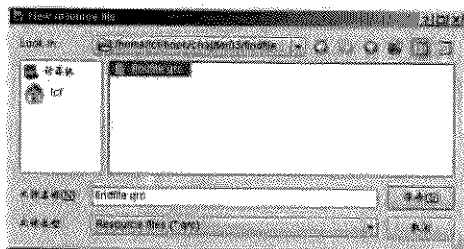


图 3-15 新建资源文件

04 在“资源编辑器”对话框中单击“+”操作，添加一个前缀，可以根据自己的需要修改，在此采用默认值，如图 3-16 所示。

05 单击“Add Files...”按钮，打开“Open File”对话框，选择需要添加的图标资源，并单击“打开”按钮，如图 3-17 所示。

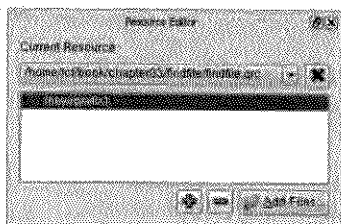


图 3-16 编辑资源文件

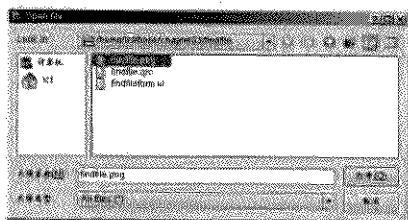


图 3-17 为资源文件打开一个资源

06 现在 findfile.png 图标文件就加入到了资源文件 findfile.qrc 中，其添加资源后的状态如图 3-18 所示。

07 在对象监视器中选择顶层窗口部件 FindFileForm 对象，并在属性编辑器中选择“windowTitle”属性，将“Form”修改为“查找文件”；选择“windowIcon”属性选项，并单击下拉框右侧的“打开”按钮，如图 3-19 所示。

08 在“Find icon”对话框中有两个单选选项：选项“Specify image file”表示通过绝对路径来定位图标文件；选项“Specify resource”表示通过资源文件来定位资源。笔者推荐使用选项“Specify resource”，因为采用绝对路径来定位图标文件，往往会导致应用程序在安装后系统无法找到该图标文件。选中“Specify resource”，这时候“Current Resource”列表中会列出已经创建的资源文件以及在文

件中添加的资源。选中刚刚添加的图标资源 findfile.png，单击“确定”按钮。如图 3-20 所示。

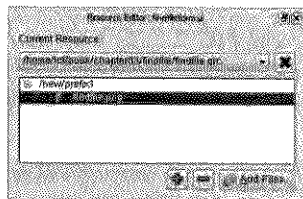


图 3-18 添加资源后的资源文件状态

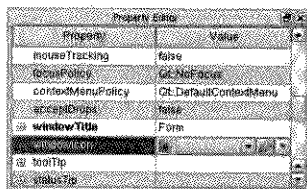


图 3-19 设置窗口部件的窗口图标

09 在 Qt 设计器中预览 GUI 用户界面（可以通过快捷键“Ctrl+R”激活预览命令），可以看到图标已经添加到顶层窗口部件“FindFileForm”用户界面的标题栏上了。

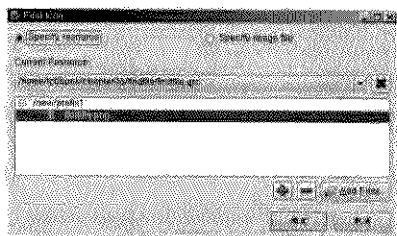


图 3-20 指定图标资源

绘制好 GUI 用户界面后，保存为“findfileform.ui”文件（该 ui 文件在源代码中的路径为“chapter03/findfile/findfileform.ui”），该文件是 XML 格式的文本文档。一个比较小的 ui 文件的内容如下所示。

```
<ui version="4.0" >
  <class>Dialog</class>
  <widget class="QDialog" name="Dialog" >
    <property name="geometry" >
      <rect>
        <x>0</x>
        <y>0</y>
        <width>400</width>
        <height>300</height>
      </rect>
    </property>
    <property name="windowTitle" >
      <string>Dialog</string>
    </property>
    <widget class="QPushButton" name="pushButton" >
      <property name="geometry" >
        <rect>
          .....
        </rect>
      </property>
    </widget>
  </widget>
</ui>
```

```

        </property>
        .....
    </widget>
    .....
</widget>
<resources/>
<connections/>
</ui>

```

有时候,在 Qt 设计器中修改窗口部件的类型和属性是比较麻烦的,比如修改顶层窗口部件的类型(例如,将 QDialog 改为 QWidget)。而直接修改 ui 文件很容易就做到了(在 Qt 设计器中无法做到,必须要重新建立一个新的 QDialog 顶层窗口部件)。有兴趣的读者可以对照 Qt 设计器中的界面和属性,对 Qt 的 ui 文件仔细分析一下,在此不再赘述。

在 Qt 设计器中绘制好 GUI 界面后,就可以利用 Qt 提供的 qmake 工具和 uic 编译工具将 ui 文件编译生成 C++ 源文件。该源文件默认的命名规则是:

```
ui_<ui 文件名>.h
```

对于绘制的 GUI 用户界面文件 findfileform.ui, uic 编译工具将生成名为“ui_findfileform.h”的 C++ 代码文件。通过在 qmake 工程文件中配置 FORMS 选项,可以让 uic 编辑工具自动完成上述过程(在下一节进行详细描述)。

现在在控制台上执行 uic 命令,看一下生成的 ui_findfileform.h 文件(也可以生成自己喜欢的名字或直接输出在控制台):

```
uic -o ui_findfileform.h findfileform.ui 或者
```

```
uic findfileform.ui (直接输出到控制台)
```

生成的 C++ 源文件的内容如下所示(由于文件过大,部分省略)。

```

/*****
** Form generated from reading ui file 'findfileform.ui'
**
** Created: Mon Aug 6 10:49:53 2007
**    by: Qt User Interface Compiler version 4.3.0
**
** WARNING! All changes made in this file will be lost when
** recompiling ui file!
*****/

#ifndef UI_FINDFILEFORM_H
#define UI_FINDFILEFORM_H

#include <QtCore/QVariant>
#include <QtGui/QAction>
#include <QtGui/QApplication>
.....
class Ui_FindFileForm
{
public:

```



```
QVBoxLayout *vboxLayout;
QGridLayout *gridLayout;
QLineEdit *txtLineEdit;
QPushButton *browsePushBtn;

.....

void setupUi(QWidget *FindFileForm)
{
    FindFileForm->setObjectName(QString::fromUtf8("FindFileForm"));
    vboxLayout = new QVBoxLayout(FindFileForm);
    vboxLayout->setSpacing(6);
    vboxLayout->setMargin(9);
    vboxLayout->setObjectName(QString::fromUtf8("vboxLayout"));
    gridLayout = new QGridLayout();
    gridLayout->setSpacing(6);
    gridLayout->setMargin(0);
    gridLayout->setObjectName(QString::fromUtf8("gridLayout"));
    txtLineEdit = new QLineEdit(FindFileForm);
    txtLineEdit->setObjectName(QString::fromUtf8("txtLineEdit"));

    gridLayout->addWidget(txtLineEdit, 1, 1, 1, 2);

    .....

    QSize size(438, 387);
    size = size.expandedTo(FindFileForm->minimumSizeHint());
    FindFileForm->resize(size);

    QMetaObject::connectSlotsByName(FindFileForm);
} // setupUi

void retranslateUi(QWidget *FindFileForm)
{
    FindFileForm->setWindowTitle(
        QApplication::translate("FindFileForm",
            "Form", 0, QApplication::UnicodeUTF8));

    .....

    Q_UNUSED();
} // retranslateUi
};

namespace Ui {
    class FindFileForm: public Ui_FindFileForm {};
} // namespace Ui

#endif // UI_FINDFILEFORM_H
```

该 `ui_findfileform.h` 文件定义了一个 POD (Plain Old Data) 类 `Ui_FindFileForm`，该类的名字来自 Qt 设计器绘制的 Qt GUI 用户界面的顶层窗口部件 (“objectName” 属性为 “FindFileForm” 的窗口部件) 的

对象名。在类 `Ui_FindFileForm` 定义的公共区，包含了在 Qt 设计器中添加的所有子窗口部件的指针，并定义了几个内联函数。其中，最重要的是公共函数 `setupUi()`，该函数对传入的 Qt 标准窗口部件 `QWidget` 对象的标题、大小等进行设置。它的参数类型“`QWidget *`”决定了使用的顶层窗口部件（或主容器）的类型是 `QWidget`，如果在 Qt 设计器中使用 `QMainWindow` 作为顶层窗口部件，那么 `setupUi()` 的参数类型将是“`QMainWindow *`”。

静态函数 `QObject::connectSlotsByName(QObject * object)` 将递归地搜寻传入的 Qt 对象 `object` 的所有子对象，并把所有匹配的子对象的信号关联到 `object` 对象的符合下列规则的槽函数（将在本章的 3.3 节详细描述）：

```
void on_<窗口部件名称>_<信号名称>(<信号参数>)
```

`setObjectName()` 函数设置对象的名称，以便能通过 `QObject::findChild()` 以及 `QObject::objectName()` 函数获取这些对象的引用。

注意，`uic` 工具生成的 `Ui_FindFileForm` 类并不是一个 `QWidget` 类，也不是一个 `QObject` 对象，它的作用仅仅是构建和部署 Qt 设计器中绘制的窗口部件。

3.2 程序中引入自定义窗口部件

保存好 Qt 设计器绘制的 GUI 界面文件后，就可以将 `findfileform.ui` 加入到 `qmake` 工程文件中，并可以在应用程序中使用该 GUI 用户界面。Qt 提供了 3 种方法在应用程序中使用 Qt 设计器绘制的界面类：

- 直接使用方式；
- 单一继承方式；
- 多继承方式。

下面一一进行详细阐述。

3.2.1 直接使用方式

首先，在“chapter03/findfile”目录下新建名字为“direct”的 KDevelop 工程。然后将 `ui` 文件加入到 `qmake` 工程中。步骤如下：

01 在 KDevelop 中，选择右侧的“QMake 管理器”选项卡，在下面的窗口中右键单击“FORMS”选项，在弹出的上下文菜单中选择“添加已有的文件...”命令，如图 3-21 所示。

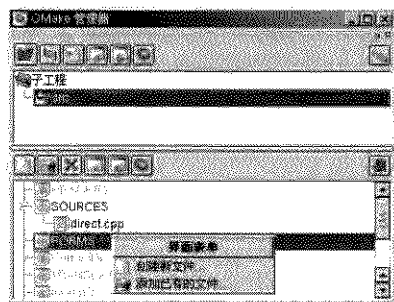


图 3-21 在 KDevelop 中为工程添加界面表单



02 在打开的对话框中，选中需要添加的文件“findfileform.ui”。在最下面的下拉框中，有三个选项：

- **复制文件**——复制 ui 文件到 qmake 工程文件 src.pro 所在的目录，并将复制的 ui 文件添加到 qmake 工程文件中；
- **添加符号连接**——在 qmake 的工程文件 src.pro 所在的目录建立一个链接，指向需要添加的 ui 文件，并将链接添加到 qmake 工程文件中；
- **添加相对路径**——以相对路径的形式添加到 qmake 工程文件 src.pro 中。

在此，选择“添加相对路径”选项，单击“确定”按钮，如图 3-22 所示。

这时候，findfileform.ui 文件便添加到 KDevelop 工程当中，如图 3-23 所示。



图 3-22 指定界面表单



图 3-23 添加界面表单后的 KDevelop 显示的 qmake 工程选项

相应地，KDevelop 会修改当前 KDevelop 工程的 src 目录下的 qmake 工程文件 src.pro，其内容为：

```
# File generated by kdevelop's qmake manager.
# -----
# Subdir relative project main directory: ./src
# Target is ????? ../bin/direct

FORMS += ../../findfileform.ui
SOURCES += direct.cpp
TARGET=../bin/direct
```

文件中以“#”开头的是 KDevelop 自动添加的一些注释和说明。这些说明指出了该 qmake 工程文件的创建者等信息。

变量 FORMS 告诉 qmake 编译工具添加了一个 findfileform.ui 文件，位置在 chapter03/findfile 目录下（注意，qmake 工程文件 src.pro 在目录 chapter03 /findfile/direct/src 下）。

此外，也可以手动修改 qmake 工程文件 src.pro，将 findfileform.ui 加入到工程中。

现在进一步修改 qmake 工程文件 src.pro，将资源文件引入到工程文件中。最终 src.pro 的内容如下。

```
FORMS += ../../findfileform.ui
SOURCES += direct.cpp
TARGET=../bin/direct
RESOURCES += ../../findfile.qrc
```

修改主程序文件 direct.cpp，其内容如下。

```
// chapter03/Findfile/direct/src/direct.cpp.
```



```

#include <QtGui>
#include <QtCore/QStringCodec>

#include "ui_findfileform.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QStringCodec::setCodecForTr(QStringCodec::codecForName("gb2312"));
    Q_INIT_RESOURCE(findfile);

    QWidget *pWidget = new QWidget;
    Ui::FindFileForm ui;
    ui.setupUi(pWidget);
    pWidget->show();

    return app.exec();
}

```

main 函数中创建了一个 Qt 标准的 QWidget 对象 pWidget。pWidget 对象用来加载(host) findfileform.ui 文件描述的 GUI 窗口部件。

Ui::FindFileForm 类来自 ui_findfileform.h 文件, 该类定义了 Qt 设计器中绘制的所有窗口部件以及关联的信号和槽。通过调用它的 setupUi() 函数, 初始化、部署窗口部件, 并设置窗体的大小和标题等。

现在编译运行应用程序。不过由于没有实现查找功能, 用户是无法同界面进行交互的。

这种在程序中使用 Qt 设计器的*.ui 文件的方式, 直接简单, 实现快。然而, 在较大型的项目中, 仅仅使用 Qt 设计器绘制用户界面是不够的, 还需要通过应用程序的代码操作控制生成的窗口部件, 达到与用户交互的目的。这就需要使用下面两种方式在应用程序中引入 Qt 设计器的 ui 文件。

3.2.2 单一继承方式

在直接使用方式中, 主程序创建了一个 QWidget 对象, 并使用 Ui_FindFileForm 界面对象的 setupUi() 函数进行初始化和加载窗口部件。而在单一继承方式中, 需要声明一个继承自 QWidget 的自定义类, 并在这个类的构造函数中初始化和加载 GUI 用户界面, 如图 3-24 所示。

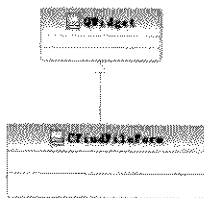


图 3-24 单一继承方式使用 Ui 类

在 chapter03/findfile 目录下, 新建 KDevelop 工程 “singleinherit”, 并新建两个文件 findfileform.h 和 findfileform.cpp, 加入到工程中。

自定义类 CFindFileForm 的头文件 findfileform.h 代码清单如下。

```
// chapter03/findfile/singleinherit/src/findfileform.h
#ifndef _FINDFILEFORM_H_
#define _FINDFILEFORM_H_

#include "ui_findfileform.h"

class CFindFileForm : public QWidget
{
    Q_OBJECT
public:
    CFindFileForm(QWidget* = 0);

private:
    Ui_FindFileForm ui;
};
#endif
```

在 CFindFileForm 类的定义文件中，定义了一个私有成员变量 ui，它的类型是 Ui_FindFileForm。前面已经讲过，类 Ui_FindFileForm 是由 uic 工具根据 findfileform.ui 文件创建的，它的功能是在构造函数中初始化、加载和管理 GUI 用户界面。

自定义类 CFindFileForm 的实现文件 findfileform.cpp 的内容如下所示。

```
// chapter03/findfile/singleinherit/src/findfileform.cpp
#include <QtGui>
#include "findfileform.h"

CFindFileForm::CFindFileForm(QWidget* parent)
: QWidget(parent)
{
    ui.setupUi(this);
    ui.statusLabel->setText(tr("就绪"));
    ui.resultLabel->setText(tr("找到 0 个文件"));
    ui.nameComboBox->setEditText("");
    ui.dirComboBox->setEditText(QDir::currentPath());
    ui.dirComboBox->addItem(QDir::currentPath());
    ui.sensitiveCheckBox->setEnabled(false);
    ui.stopPushBtn->setEnabled(false);
}
```

在类 CFindFileForm 的构造函数中调用了它的成员变量 ui 的 setupUi() 函数，对 Qt 设计器绘制的 GUI 用户界面进行初始化和加载。

然后，程序对用户界面中的一些窗口部件的状态进行了初始化。QComboBox::setEditText() 设置下拉列表框的显示文本，该文本对用户是可编辑的。静态函数 QDir::currentPath() 返回应用程序所在目录的绝对路径。

QCheckBox::setEnabled(false) 设置“区分大小写”复选框的初始状态为不可用的（显示为灰色）。

QPushButton::setEnabled(false) 设置“停止查找”按钮为不可用的。

现在，在主函数 main() 中实例化和显示 Qt 设计器绘制的 GUI 用户界面。singleinherit.cpp 文件的

内容如下。

```
// chapter03/findfile/singleinherit/src/singleinherit.cpp.
#include <QtGui/QApplication>
#include <QtCore/QTextCodec>
#include "findfileform.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForName("gb2312"));
    Q_INIT_RESOURCE(findfile);

    CFindFileForm form;
    form.show();

    return app.exec();
}
```

主程序很简单，首先声明了一个 CFindFileForm 对象 form，然后调用该对象的公共函数 show() 进行显示。

为了简便起见，此处没有实现真正的查找文件功能，只是显示界面，而没有响应。单一继承方式使用 Qt 设计器绘制的用户界面是能够实现同用户进行交互的，它和下一节讲到的“多继承方式”基本相同。不同的是，单一继承方式需要通过成员 ui 来引用 Qt 设计器中部署的窗口部件，而在多继承方式中可以直接引用这些窗口部件。在下一小节中，将定义信号和槽，实现文件的查找功能。

将 findfileform.ui 文件和资源文件 findfile.qrc 加入到 qmake 工程文件 src.pro 中，src.pro 文件的内容如下：

```
FORMS += ../../findfileform.ui
HEADERS += findfileform.h
SOURCES += singleinherit.cpp \
           findfileform.cpp
TARGET=../bin/singleinherit
RESOURCES += ../../findfile.qrc
```

编译运行应用程序。

单一继承方式将 Qt 设计器绘制的 GUI 用户界面对象包含在自定义类中，作为一个私有成员来使用。这种方式的优点是，应用程序能够控制用户界面的外观和显示方式，并能够同用户进行交互；此外，还可以使用同样的一个 ui 文件来生成多个不同的自定义界面类，可以重复使用 Qt 设计器绘制的 GUI 用户界面。

3.2.3 多继承方式

多继承方式中，需要从标准的 QWidget 类和 Qt 设计器绘制的界面类继承。这样，继承自 Qt 标准的窗口部件类和界面类的子类就可以直接访问其父类的公有成员和保护成员，如图 3-25 所示。

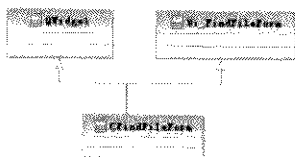


图 3-25 多继承关系

在 chapter03 /findfile 目录下，新建 KDevelop 工程“mulinherit”。新建两个文件 findfileform.h 和 findfileform.cpp，并加入到工程中。

多继承方式中，自定义类 CFindFileForm 的定义如下。

```
// chapter03/findfile/mulinherit/src/findfileform.h
#ifndef _FINDFILEFORM_H_
#define _FINDFILEFORM_H_

#include <QStringList>
#include <QDir>

#include "ui_findfileform.h"

class CFindFileForm : public QWidget,
                     public Ui_FindFileForm
{
    Q_OBJECT
public:
    CFindFileForm(QWidget* = 0);
};

#endif
```

类 CFindFileForm 以公有方式继承自 QWidget 和 Ui_FindFileForm。

注意，多继承的情况下，Qt 的类必须要放在其他类的前面，即必须先继承 QWidget，再继承 Ui_FindFileForm。这是因为元对象编译器（Meta-Object Compiler, moc）假定第一个继承的类是 QObject 的子类。此外，Qt 不支持对 QObject（或 QObject 子类）的虚继承（virtual inheritance）。

类 CFindFileForm 的实现文件 findfileform.cpp 内容如下。

```
// chapter03/findfile/mulinherit/src/findfileform.cpp
#include <QtGui>

#include "findfileform.h"

CFindFileForm::CFindFileForm(QWidget* parent)
    : QWidget(parent)
{
    setupUi(this);
    statusLabel->setText(tr("就绪"));
    resultLabel->setText(tr("找到 0 个文件"));
}
```

```

nameComboBox->setEditText("");
dirComboBox->setEditText(QDir::currentPath());
dirComboBox->addItem(QDir::currentPath());
sensitiveCheckBox->setEnabled(false);
stopPushBtn->setEnabled(false);
}

```

在类 CFindFileForm 的构造函数中，直接调用了父类 Ui_FindFileForm 的函数 setupUi()，来进行界面窗口部件的初始化、加载和部署。

接着，程序初始化了各子窗口部件的初始状态。与单一继承方式使用 ui 文件不同的是，多继承方式下可以在类的实现代码中直接引用 Qt 设计器绘制的子窗口部件。

将 findfileform.ui 文件和资源文件 findfile.qrc 加入到 qmake 工程文件 src.pro 中，src.pro 文件的内容如下。

```

FORMS += ../../findfileform.ui
HEADERS += findfileform.h
SOURCES += mulinherit.cpp \
           findfileform.cpp
TARGET=../bin/mulinherit
RESOURCES += ../../findfile.qrc

```

编译并运行应用程序。

在应用程序中，多继承方式使用 Qt 设计器中绘制的界面窗口部件比单一继承方式更简单直接；同直接使用方式相比较，多继承方式更具代码的可扩展性。因此，在应用程序中，一般的都使用多继承方式引用 Qt 设计器绘制的用户界面对象。在下一节中，讲述 Qt 的信号和槽的时候，就是通过多继承方式来实现查找文件功能的。

3.3 Qt 的信号和槽机制

在前面章节的学习中，已经初步使用了 Qt 的信号和槽，对 Qt 的信号和槽机制有了一个基本的认识。在此基础上，现在就 Qt 的信号和槽机制进行深入的阐述。

3.3.1 基本原理

在 GUI 用户界面中，当用户操作一个窗口部件时，需要其他窗口部件的响应或者能够激活其他的操作。在程序开发中，经常使用回调（callback）机制来实现。所谓回调，就是事先将一个回调函数（callback function）指针传递给某一个处理过程，当这个处理过程得到执行时，回调预先定义好的回调函数以期实现激活其他处理过程的目的。

不同于回调函数机制，Qt 提供了信号和槽机制。信号是一个特定的标识；一个槽就是一个函数，与一般的函数不同，槽函数既能够和信号关联，也能够像普通函数一样直接调用。当某个事件出现时，通过发送信号，可以将与之相关联的槽函数激活，即执行槽函数代码。在程序中，使用 QObject::connect() 函数来将某个信号和某个槽进行关联，而信号和槽之间的真正关联是由 Qt 的信号和槽机制来实现的。

信号和槽的关联关系可以有几种模式：

- 一个信号和一个槽关联；
- 一个信号和多个槽关联；



- 多个信号和一个槽关联。

一个信号和多个槽关联的情况下，当发出该信号的时候，与该信号关联的各个槽以任意的先后顺序立即执行（即槽函数的执行顺序是随机的，与槽关联的顺序没有关系）。信号和槽机制是完全独立于 GUI 事件循环的。

此外，信号还可以和信号进行关联。当两个信号关联时，第一个信号的发出将会激活 Qt 发送第二个信号。

信号和槽通过 `QObject::connect (const QObject * sender, const char * signal, const QObject * receiver, const char * method, Qt::ConnectionType type = Qt::AutoCompatConnection)` 函数关联，参数 `type` 定义了信号和槽的关联方式，决定一个信号是立即传递到槽还是排队等待以后传递。Qt 使用枚举类型 `Qt::ConnectionType` 定义信号和槽的关联方式，有三种：

- `Qt::DirectConnection`，信号发送后立即传递给相关联的槽。只有槽函数执行完毕返回后，发送信号 “emit <信号>” 之后的代码才被执行。
- `Qt::QueuedConnection`，信号发送后排队，直到事件循环（event loop）有能力将它传递给槽；而不管槽函数有没有执行，发送信号 “emit <信号>” 之后的代码都会立即得到执行。
- `Qt::AutoConnection`：如果信号和槽在同一个线程，信号发出后，槽函数将立即执行，等同于 `Qt::DirectConnection`；如果信号和槽不在同一个线程，信号将排队，等待事件循环的处理，效果等同于 `Qt::QueuedConnection`。

对于 `Qt::QueuedConnection` 方式，Qt 元对象系统（meta-object system）必须知道信号/槽的参数类型，否则的话编译器会报错 “`QObject::connect: Cannot queue arguments of type 'Type'.`”，要想使用类型 `Type` 必须首先通过 `qRegisterMetType()` 函数在元对象系统中注册（简单数据类型和 Qt 定义的数据类型读者无需注册，可以直接使用）。对于不同线程间的通信，将在后面的章节中介绍。

在第 1 章提到过，宏 `SIGNAL()` 和 `SLOT()` 返回其参数的 C 风格的字符串（`const char *`），因此下面关联信号和槽的两个语句是等价的：

```
connect(pushButton, SIGNAL(clicked()), this, SLOT(doPushButton()));
connect(pushButton, "clicked()", this, "doPushButton());
```

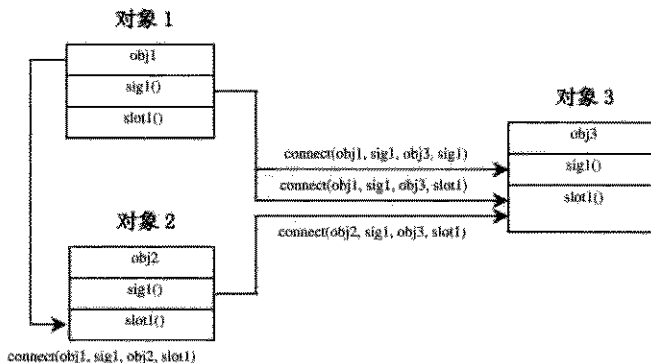


图 3-26 信号和槽的关联

Qt 信号和槽机制的优点是：

- **类型安全的。**需要关联的信号和槽的签名必须是等同的，即信号的参数类型和参数个数同接受

该信号的槽的参数类型和参数个数相同；不过，一个槽的参数个数是可以少于信号的参数个数的，但缺少的参数必须是信号参数的最后一个或几个参数。如果信号和槽的签名不符，编译器就会报错。

- **松散耦合的。**Qt 信号和槽机制减弱了 Qt 对象的耦合度。激发信号的 Qt 对象无须知道是哪个对象的哪个槽需要接收它发出的信号，它只需要做的是在适当的时间发送适当的信号就可以了，而不需要知道也不关心它的信号有没有被接收到，更不需要知道是哪个对象的哪个槽接收到了信号；同样的，对象的槽也不知道是哪些信号关联到了自己。而一旦关联信号和槽，Qt 就保证了适合的槽得到调用。即使关联的对象在运行时被删除，应用程序也不会出现崩溃。

一个类要想支持信号和槽，必须从 QObject 或 QObject 的子类继承。注意，Qt 信号和槽机制不支持对模板的使用。



扩展阅读

关于信号/槽机制的效率问题

信号和槽机制增强了对对象间通信的灵活性，然而这也损失了一些性能。同回调函数相比较，信号和槽机制是有些慢的。通常，通过传递一个信号来调用槽函数将会比直接调用非虚函数慢 10 倍。原因主要有：

- 需要定位接收信号的对象；
- 安全地遍历所有的关联（比如，一个信号关联到多个槽的情况）；
- 编组（marshal）/解组（unmarshal）传递的参数；
- 多线程的时候，信号可能需要排队等待。

然而，与创建堆对象的 new 操作以及删除堆对象的 delete 操作相比较，信号和槽的代价只是它们很少的一部分。信号和槽机制导致的这点性能损耗，对实时应用程序是可以忽略的。同信号和槽提供的灵活性和简便性相比，这点性能的损失也是值得的。

3.3.2 设计信号和槽

在学习了 Qt 信号和槽的基本机制后，现在设计槽函数来响应用户的查找文件操作。

下面，以多继承方式使用 Qt 设计器绘制的界面类，实现文件的查找功能。加入自定义槽后的类 CFindFileForm，其定义如下所示。

```
// chapter03/findfile/mulinherit/src/findfileform.h
#ifndef _FINDFILEFORM_H_
#define _FINDFILEFORM_H_

#include <QStringList>
#include <QDir>
#include "ui_findfileform.h"

class CFindFileForm : public QWidget,
                     public Ui_FindFileForm
{
```

```

    Q_OBJECT
public:
    CFindFileForm(QWidget* = 0);

private:
    QStringList findFiles(const QDir&, const QString&, const QString&);
    void showFiles(const QDir&, const QStringList&);
    void transFolder(const QDir&, const QString&, const QString);
    bool m_bStoped;
    bool m_bSubfolder;
    bool m_bSensitive;
    int m_nCount;

private slots:
    void browse();
    void find();
    void stop();
    void doTxtChange(const QString&);
};
#endif

```

头文件中，包含了头文件<QStringList>，该文件包含了需要的类 QString 和 QStringList 的定义。头文件<QDir>包含了类 QDir 的定义，类 QDir 提供了对目录结构和目录的访问方法。

在 CFindFileForm 类的私有区，声明了必需的成员函数和成员变量：

- findFiles() 函数实现文件的查找并返回符合条件的文件列表；
- transFolder() 是一个递归函数，实现对文件夹的递归查找。
- 成员变量 m_bStoped 记录用户是否单击了“停止”按钮，如果是，m_bStoped 为 true，否则为 false。
- 成员变量 m_bSubfolder 记录用户是否需要查找子文件夹中的文件，如果需要查找则 m_bSubfolder 为 true，否则为 false。
- 成员变量 m_bSensitive 用于查找包含特定文本信息的文件时，是否区分文本信息的大小写。如果用户选中了“区分大小写”复选框，则 m_bSensitive 为 true，否则为 false；
- m_nCount 记录查找到文件的总数。

在“private slots”区，声明了四个槽。其中，browse() 槽函数响应用户的单击“浏览...”操作；find() 槽函数响应用户的单击“查找”操作；stop() 槽函数响应用户的“停止查找”操作；doTxtChange() 槽函数响应用户的输入“包含文本”操作。所有的四个槽函数返回类型都是 void。当槽函数响应信号而执行时，槽函数的返回类型被忽略掉。但是槽函数作为一般函数使用时（即直接在程序中调用槽函数），槽函数的返回类型是有用的。在这种情况下，声明返回类型为非 void 的槽函数是必要的。

接下来，看一下类 CFindFileForm 的实现文件 findfileform.cpp，它包含了头文件中定义的所有函数和槽函数的实现。

```

// chapter03/findfile/mulinherit/src/findfileform.cpp
#include <QtGui>
#include "findfileform.h"

```



```

CFindFileForm::CFindFileForm(QWidget* parent)
:   QWidget(parent),
    m_bStopped(false),
    m_nCount(0)
{
    setupUi(this);
    statusLabel->setText(tr("就绪"));
    resultLabel->setText(tr("找到 0 个文件"));
    nameComboBox->setEditText("");
    dirComboBox->setEditText(QDir::currentPath());
    dirComboBox->addItem(QDir::currentPath());
    sensitiveCheckBox->setEnabled(false);

    connect(findPushBtn, SIGNAL(clicked()), this, SLOT(find()));
    connect(stopPushBtn, SIGNAL(clicked()), this, SLOT(stop()));
    connect(closePushBtn, SIGNAL(clicked()), this, SLOT(close()));
    connect(browsePushBtn, SIGNAL(clicked()), this, SLOT(browse()));
    connect(txtLineEdit, SIGNAL(textChanged(const QString&)),
            this, SLOT(doTxtChange(const QString&)));
}

```

在类 CFindFileForm 的构造函数中，首先在它的初始化列表中对成员变量进行了必要的初始化。接着，初始化 Qt 窗口部件的初始状态。最后，将 Qt 窗口部件的信号关联到相应的槽。在行编辑框 QLineEdit 的文本发生改变的时候，将会发送 QLineEdit::textChanged(const QString& text) 信号，该信号的参数 text 是用户输入到行编辑框的新文本。

现在，看一下响应用户“浏览...”操作的槽函数 browse()。

```

void CFindFileForm::browse()
{
    QString dir = QFileDialog::getExistingDirectory(this,
                                                    tr("选择查找路径"),
                                                    QDir::currentPath(),
                                                    QFileDialog::ShowDirsOnly);

    if (!dir.isEmpty())
    {
        dirComboBox->addItem(dir);
        dirComboBox->setCurrentIndex(dirComboBox->currentIndex() + 1);
    }
}

```

函数 QFileDialog::getExistingDirectory() 打开一个文件对话框，它将返回用户选择的文件系统中存在的路径。静态函数 QDir::currentPath() 获取应用程序所在的路径，并初始化为文件对话框的当前目录。实参 QFileDialog::ShowDirsOnly 指示文件对话框只显示目录。

在文件对话框返回的路径非空的情况下，QComboBox::addItem() 将返回的路径添加到“查找位置”下拉框 dirComboBox 中，并通过 QComboBox::setCurrentIndex() 设置为下拉框的当前显示内容。

```

void CFindFileForm::find()
{

```



```
frame->setEnabled(false);
findPushBtn->setEnabled(false);
stopPushBtn->setEnabled(true);
statusLabel->setText(tr("正在搜索..."));
resultTableWidget->setRowCount(0);

QString fileName = nameComboBox->currentText();
QString txt = txtLineEdit->text();
QString path = dirComboBox->currentText();
m_bSubfolder = subfolderCheckBox->isChecked();
m_bSensitive = sensitiveCheckBox->isChecked();

m_nCount = 0;
m_bStoped = false;
QDir dir = QDir(path);
if (fileName.isEmpty())
    fileName = "***";
travFolder(dir, fileName, txt);

if(m_bStoped)
    statusLabel->setText(tr("已中止"));
else
    statusLabel->setText(tr("就绪"));
findPushBtn->setEnabled(true);
stopPushBtn->setEnabled(false);
frame->setEnabled(true);
}
```

CFindFileForm::find()槽函数响应用户的单击“查找”操作，实现查找文件的功能。

当用户单击“查找”按钮的时候，首先改变显示窗口部件的状态。容器 frame 变为不可用时，它的所有子窗口部件的状态将会成为不可用的。这样做是为了防止应用程序在进行文件查找的过程中用户再次进行输入操作。这也是使用 QFrame 容器的主要目的。当“查找”按钮变为不可用时，“停止查找”按钮变为可用，允许查找的过程中用户中断查找。接着设置“状态”标签的显示内容为“正在搜索...”以提示用户已经进入查找，同时清理显示查找结果的表格，设置它的行数为 0。

接着，取得要查找的文件名字、包含的文本和查找的路径并保存在相应的变量中：保存“区分大小写”和“含子文件夹”复选框的选择状态。同时将记录查找文件数目的成员变量 m_nCount 清零，成员变量 m_bStoped 设置为 false，以备查找过程中使用。

查找之前需要获取用户选择的查找路径，因此调用了 QDir 类的构造函数来构造 QDir 对象 dir，它指向用户选择的目录。

最后判断用户输入的要查找的文件名是否为空。如果用户没有任何输入，那么应用程序将查找有文件，即将目录下所有的文件列出显示在表格中。

函数 travFolder()实现真正的查找文件功能。它以用户选择的查找位置、查找的文件名和文件中包含的文本作为输入参数，将最终的查找结果显示在表格中。

在完成查找并显示查找结果后，窗口部件将恢复到查找之前的状态，以便用户能够重新输入查找条件，执行下一个查找操作。

```

void CFindFileForm::tranvFolder(const QDir& dir,
                                const QString& fileName,
                                const QString txt)
{
    if(m_bSubfolder)
    {
        QStringList folders;
        folders = dir.entryList(QDir::Dirs | QDir::NoDotAndDotDot);
        for (int i = 0; i < folders.size(); ++i)
        {
            QApplication->processEvents();
            if (m_bStopped)
                break;

            QString strDir =
                QString("%1/%2").arg(dir.absolutePath())
                    .arg(folders[i]);
            tranvFolder(strDir, fileName, txt);
        }
    }
    QStringList files = findFiles(dir, fileName, txt);
    showFiles(dir, files);
    m_nCount += files.size();
    resultLabel->setText(tr("找到%1 个文件").arg(m_nCount));
}

```

tranvFolder()是一个递归函数，它递归地实现文件的查找（如果用户选择了“包含子文件夹”复选框的话）并将查找结果显示在用户界面上。该函数具有三个形参，分别用来接受用户输入的查找路径、文件名和包含的文本。

首先，该函数判断用户是否选中了“包含子文件夹”复选框，如果选中，该函数将会执行文件的递归查找。在递归查找文件的情况下，QDir::entryList()函数获取当前文件夹下的所有子文件夹的列表并保存在 QStringList 对象 folders 中。QDir::entryList()函数具有一个实参，它指定了该函数的过滤器，QDir::Dir 表示只获取当前路径下的文件夹；QDir::NoDotAndDotDot 表示获取的文件夹列表中不包含“.”（当前目录）和“..”（当前目录的上一级目录）目录。这样的做的目的很简单，防止应用程序永远在当前目录和上一级目录中循环，无法退出。接着应用程序递归的遍历当前目录的子目录列表，进行文件的查找。在这个过程中，调用了 QApplication::processEvents()事件处理函数（关于事件处理的内容，将在第 11 章详细介绍），以便应用程序即使在查找过程中也能够响应用户的“停止查找”或“关闭”操作。最后获取子目录的绝对路径，并将它以及文件名和包含文本传递给递归函数 tranvFolder()，进行进一步的查找。

如果用户没有选中“包含子文件夹”复选框或者递归函数已经遍历完了当前目录下的所有子目录，那么 tranvFolder() 函数进入对当前文件夹下的文件查找。findFiles() 函数接受当前目录的绝对路径、文件名和包含文本作为参数，实现对当前目录的文件查找，并返回查找到的文件列表。showFiles() 函数进行文件的显示。最后在用户界面上显示当前应用程序查找到的符合条件的文件个数。

函数 QString::arg()用传递给它的参数代替原字符串中对应的“%1”，以构建一个新的字符串并返



回（详见第 13 章的 13.2 节）。

```
QStringList CFindFileForm::findFiles(const QDir &dir,
    const QString &fileName,
    const QString &txt)
{
    QStringList files = dir.entryList(QStringList(fileName),
        QDir::Files | QDir::NoSymLinks);
    if (txt.isEmpty())
        return files;
    QStringList foundFiles;
    Qt::CaseSensitivity sensitive = Qt::CaseInsensitive;
    if (m_bSensitive)
        sensitive = Qt::CaseSensitive;
```

函数 `findFiles()` 对一个特定的目录实现文件的查找，它具有和 `travFolder()` 函数相同的参数个数和参数类型，但它返回的是一个查找结果的文件列表。

函数 `QDir::entryList()` 获取具有给定文件名的文件列表。它具有两个实参，第一个参数是一个 `QStringList` 类型的列表，指定了一个文件名过滤器；第二个参数指定了另一个过滤器，`QDir::Files` 表示只获取当前目录下的文件的名称（不包含目录），`QDir::NoSymLinks` 表示返回的文件名列表不含系统的符号链接。

接着判断是否区分大小写，并将状态保存下来。

```
for (int i = 0; i < files.size(); ++i)
{
    QApplication::processEvents();
    if (m_bStoped)
        break;

    QFile file(dir.absoluteFilePath(files[i]));
    if (file.open(QIODevice::ReadOnly))
    {
        QString line;
        QTextStream in(&file);
        while (!in.atEnd())
        {
            if (m_bStoped)
                break;
            line = in.readLine();
            if (line.contains(txt, sensitive))
            {
                foundFiles << files[i];
                break;
            }
        }
    }
}
return foundFiles;
}
```

对当前目录的所有符合文件名字条件的文件内容进行遍历,看是否包含用户指定的文本。如果文件符合用户的查找要求则将文件名字保存在一个列表中。最后返回查找的结果列表。在这个过程中,用户同样可以终止应用程序对文件的查找。

`QDir::absoluteFilePath()`函数返回当前目录下一个文件的绝对路径名。该函数不会检查该文件是否真正存在,它只是将给定的文件名和 `QDir` 对象具有的绝对路径构建一个新的绝对路径名。因为程序已经保证了该文件是确实存在的,所以不存在检查的问题。该函数的返回结果作为创建一个 `QFile` 对象的参数, `QFile` 对象提供了对文件的读写操作。

`QFile::open()`试着打开文件, `QIODevice::ReadOnly` 参数指定了以只读的方式打开文件。如果能够打开,则返回 `true`, 否则返回 `false`。

接着构造一个 `QTextStream` 栈对象 `in`, `QTextStream` 提供了对文本进行读写的操作。然后函数 `QTextStream::readLine()` 对文件进行行读入操作,函数 `QString::contains()`进行文本的包含判断,直到函数 `QTextStream::atEnd()`判断已经到了文件尾。

```
void CFindFileForm::showFiles(const QDir &dir, const QStringList &files)
{
    for (int i = 0; i < files.size(); ++i)
    {
        QString strFilePath = dir.absoluteFilePath(files[i]);
        QFile file(strFilePath);
        QFileInfo fileInfo(file);
        quint64 size = fileInfo.size();
        QDateTime dateTime = fileInfo.created();
        QString strDateTime = dateTime.toString(tr("yyyy MM月dd日 hh:mm"));
        QString strPermission;
        if (fileInfo.isWritable())
            strPermission = ("w");
        if (fileInfo.isReadable())
            strPermission.append(" r");
        if (fileInfo.isExecutable())
            strPermission.append(" x");

        QTableWidgetItem *fileNameItem =
            new QTableWidgetItem(strFilePath);
        fileNameItem->setFlags(Qt::ItemIsEnabled);
        QTableWidgetItem *sizeItem = new QTableWidgetItem(tr("%1 KB")
            .arg(int((size + 1023) / 1024)));
        sizeItem->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
        sizeItem->setFlags(Qt::ItemIsEnabled);
        QTableWidgetItem *createdItem = new QTableWidgetItem(strDateTime);
        QTableWidgetItem *permissionItem =
            new QTableWidgetItem(strPermission);

        int row = resultTableWidget->rowCount();
        resultTableWidget->insertRow(row);
        resultTableWidget->setItem(row, 0, fileNameItem);
        resultTableWidget->setItem(row, 1, sizeItem);
        resultTableWidget->setItem(row, 2, createdItem);
```



```

        resultTableWidget->setItem(row, 3, permissionItem);
    }
}

```

showFiles()函数在用户界面上显示查找的结果。它具有两个参数，第一个是个 QDir 类型的参数，它指定了文件所在目录的绝对路径；第二个参数是文件名列表。

QFileInfo 对象提供了系统无关的文件信息，包括文件名、文件大小、在文件系统中的位置、访问权限以及是否是一个符号链接等。在此，程序利用它获取文件的大小信息和访问权限信息。

最后，程序将获得的文件的信息显示在用户界面的表格中。

```

void CFindFileForm::stop()
{
    m_bStoped = true;
}

```

槽函数 stop()响应用户的单击“停止”按钮操作，并将用户的意图保存在成员变量 m_bStoped 中。

```

void CFindFileForm::doTxtChange(const QString& txt)
{
    if(txt.isEmpty())
        sensitiveCheckBox->setEnabled(false);
    else
        sensitiveCheckBox->setEnabled(true);
}

```

槽函数 doTxtChanged()响应“包含文本”行编辑框对象 txtLineEdit 的编辑操作，如果 txtLineEdit 的文本发生了变化，那么将会激发该槽函数的执行。它的作用是判断“包含文本”行编辑框的内容是否为空，如果为空，将“区分大小写”复选框 sensitiveCheckBox 的状态置为不可用，此时用户不能够作出是否进行“区分大小写”查找的选择；如果“包含文本”行编辑框的内容不为空，则置为可用。

现在对应用程序进行重编译，运行界面如图 3-27 所示。

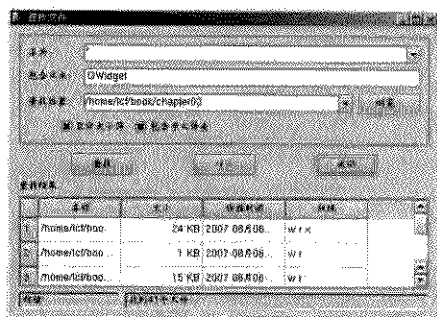


图 3-27 查找文件运行界面

3.3.3 信号和槽的自动关联

除了能够在程序中手动关联信号和槽之外，Qt 的元对象提供了信号和槽的自动关联。对于 Qt 窗

口部件已经提供的信号,如果能够按下面的规则命名槽函数,那么 Qt 就能够自动进行关联:

```
void on_<窗口部件名>_<信号名>(<信号参数>);
```

例如,对于 Qt 设计器绘制的“查找文件”窗口部件中的“浏览...”按钮 browsePushBtn,以及它的单击信号“clicked()”,将原来的槽函数 void browse() 修改为 on_browsePushBtn_clicked()。修改 findfileform.h 的声明,定义如下:

```
class CFindFileForm : public QWidget,
                     public Ui_FindFileForm
{
    Q_OBJECT
    .....
private slots:
    void on_browsePushBtn_clicked();
    .....
};
```

实现文件 findfileform.cpp 中的槽如下所示。

```
void CFindFileForm::on_browsePushBtn_clicked()
{
    .....
}
```

屏蔽 CFindFileForm 类构造函数中的“浏览...”按钮的信号和槽的关联操作:

```
CFindFileForm::CFindFileForm(QWidget* parent)
: QWidget(parent),
  m_bStoped(false),
  m_nCount(0)
{
    setupUi(this);
    .....
    // connect(browsePushBtn, SIGNAL(clicked{}), this, SLOT(browse{}));
    .....
}
```

编译运行程序,在“查找文件”对话框中单击“浏览...”按钮。OK,应用程序依然能够响应用户的操作。

在大型的 Qt 应用程序开发中,使用 Qt 信号/槽的自动关联功能,可以加速应用程序的开发进度。

3.4 窗口标志及几何布局

QWidget 是所有 Qt GUI 界面类的基类,它接受鼠标、键盘以及其他的窗口事件,并在显示器上绘制自己。

通过传入 QWidget 构造函数的参数(或者调用 QWidget::setWindowFlags()和 QWidget::setParent()函数)可以指定一个窗口部件的窗口标志(window flags)和父窗口部件。

窗口部件的窗口标志定义了窗口部件的窗口类型和窗口提示(bint)。窗口类型指定了窗口部件的

窗口系统属性 (window-system properties)，一个窗口部件只有一个窗口类型。窗口提示定义了顶层窗口的外观，一个窗口可以有多个提示 (提示能够进行按位或操作)。

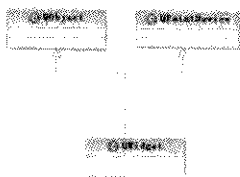


图 3-28 QWidget 类图

3.4.1 窗口标志

没有父窗口部件的 Widget 对象是一个窗口，窗口通常具有一个窗口边框 (frame) 和一个标题栏。在 Qt 4 中，QMainWindow 和所有的 QDialog 对话框子类都是经常使用的窗口类型。而子窗口部件通常处在父窗口部件的内部，没有窗口边框和标题栏。

QWidget 窗口部件的构造函数 `QWidget(QWidget* parent = 0, Qt::WindowFlags f = 0)` 具有两个形参：

- **参数 parent**，指定了窗口部件的父窗口部件，如果 `parent = 0` (默认值)，新建的窗口部件将是一个窗口；否则，新建的窗口部件是 `parent` 的子窗口部件 (是否是一个窗口还需要第二个参数决定)，如果新窗口部件不是一个窗口的话，它将会出现在父窗口部件的界面内部。
- **参数 f**，指定了新窗口部件的窗口标志，默认值是 0，即 `Qt::Widget`。

QWidget 定义的窗口类型 (为 `Qt::WindowFlags` 枚举类型，它们的可用性依赖于窗口管理器是否支持它们) 有：

- **Qt::Widget**，QWidget 构造函数的默认值。如果新的窗口部件没有父窗口部件，那么它是一个独立的窗口，否则是一个子窗口部件。
- **Qt::Window**，不管是否具有父窗口部件，新窗口部件都是一个窗口，通常具有一个窗口边框和一个标题栏。
- **Qt::Dialog**，新窗口部件是一个对话框，它是 QDialog 构造函数的默认值。
- **Qt::Sheet**，新窗口部件是一个 Macintosh 表单 (sheet)。
- **Qt::Drawer**，新窗口部件是一个 macintosh 抽屉 (drawer)。
- **Qt::Popup**，新窗口部件是一个弹出式顶层窗口。
- **Qt::Tool**，新窗口部件是一个工具 (tool) 窗口，它通常是一个用于显示工具按钮的小窗口；如果一个工具窗口具有父窗口部件，它将显示在父窗口部件的上面，否则的话，相当于使用了 `Qt::WindowStaysOnTopHint` 提示。
- **Qt::ToolTip**，新窗口部件是一个提示窗口，没有标题栏和窗口边框。
- **Qt::SplashScreen**，新窗口部件是一个欢迎窗口 (splash screen)，它是 QSplashScreen 构造函数的默认值。
- **Qt::Desktop**，新窗口部件是桌面，它是 QDesktopWidget 构造函数的默认值。
- **Qt::SubWindow**，新窗口部件是一个子窗口，而不管该窗口部件是否具有父窗口部件。

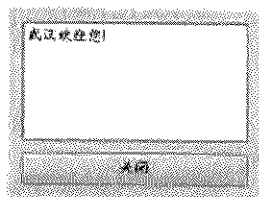


图 3-29 Qt::ToolTip 窗口

此外, Qt 定义了一些控制窗口外观的窗口提示(这些窗口提示只对顶层窗口有效):

- **Qt::MSWindowsFixedSizeDialogHint**, 为 Windows 系统上的窗口装饰一个窄的对话框边框, 通常这个提示用于固定大小的对话框。
- **Qt::MSWindowsOwnDC**, 为 Windows 系统上的窗口添加自身的显示上下文(display context)。
- **Qt::X11BypassWindowManagerHint**, 完全忽视窗口管理器, 它的作用是产生一个根本不被管理的无窗口边框的窗口(此时, 用户无法使用键盘进行输入, 除非手动调用 `QWidget::activateWindow()` 函数)。
- **Qt::FramelessWindowHint**, 产生一个无窗口边框的窗口, 此时用户无法移动该窗口和改变它的大小。
- **Qt::CustomizeWindowHint**, 关闭默认的窗口标题提示。
- **Qt::WindowTitleHint**, 为窗口装饰一个标题栏。
- **Qt::WindowSystemMenuHint**, 为窗口添加一个窗口系统菜单, 并尽可能地添加一个关闭按钮。
- **Qt::WindowMinimizeButtonHint**, 为窗口添加一个最小化按钮。
- **Qt::WindowMaximizeButtonHint**, 为窗口添加一个最大化按钮。
- **Qt::WindowMinMaxButtonsHint**, 为窗口添加一个最小化按钮和一个最大化按钮。
- **Qt::WindowContextHelpButtonHint**, 为窗口添加一个上下文帮助按钮。
- **Qt::WindowStaysOnTopHint**, 告知窗口系统该窗口应该停留在所有其他窗口的上面。
- **Qt::WindowType_Mask**, 一个用于提取窗口标志中的窗口类型部分的掩码。

枚举类型 `Qt::WindowFlags` 的低位 1 个字节用于定义窗口部件的窗口类型, 由 `0x00000000` 到 `0x00000012` 共定义了 11 个窗口类型。`Qt::WindowFlags` 的高位字节定义了窗口提示, 窗口提示能够进行位或操作, 例如,

```
Qt::WindowContextHelpButtonHint | Qt::WindowMaximizeButtonHint
```

当 `Qt::WindowFlags` 的窗口提示部分全部为 0 时, 窗口提示不会起作用。当有一个窗口提示被应用时, 要想其他的窗口提示起作用, 就必须使用位或操作(如果窗口系统支持这些窗口提示的话)。例如,

```
Qt::WindowFlags flags = Qt::Window;
widget->setWindowFlags(flags);
```

`widget` 窗口部件是一个窗口, 它具有一般窗口的外观(具有窗口边框、标题栏、最小化按钮、最大化按钮和关闭按钮等)。此时窗口提示不会起作用。

```
flags |= Qt::WindowTitleHint;
widget->setWindowFlags(flags);
```

上述代码的执行, 将会使得窗口提示发挥作用。在 Windows XP 系统上, `widget` 窗口部件是一个窗口, 它仅仅具有标题栏, 没有最小化按钮、最大化按钮和关闭按钮等。而在红旗 Linux 5.0 工作站和 SuSE 10.2 系统上, 上述代码并不起作用, 这说明 X11 窗口管理器忽略了窗口提示 `Qt::WindowTitleHint`。

在 Windows XP 系统上, 如果要添加一个最小化按钮, 必须重新设置窗口部件的窗口标志(在红旗 Linux 5.0 工作站和 SuSE 10.2 系统上, 下面的窗口提示也被忽略了),

```
flags |= Qt::WindowMinimizeButtonHint;
widget->setWindowFlags(flags);
```



如果要取消设置的窗口提示，则

```
flags &= Qt::WindowType_Mask;
widget->setWindowFlags(flags);
```

3.4.2 窗口部件的几何布局

QWidget 提供了一些处理窗口部件的几何布局的函数，可以分为两类：

- 包含窗口边框的处理函数，有 `x()`、`y()`、`frameGeometry()`、`pos()`和 `move()`。
- 不包含窗口边框的处理函数，有 `geometry()`、`width()`、`height()`、`rect()`、`size()`和 `resize()`。

Qt 窗口的几何布局如图 3-30 所示。

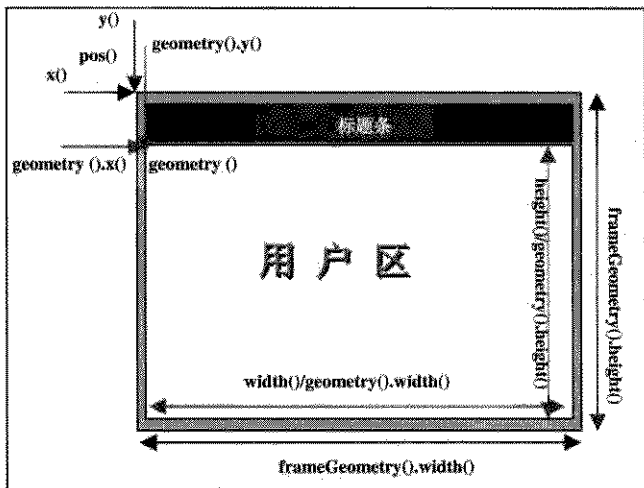


图 3-30 Qt 窗口的几何布局

Linux 采用 X11 窗口系统，它是不同于 Windows 系统的一种用户界面技术。因此，在 Linux 系统上使用 QWidget 的这些函数时，常常出现一些令人迷惑的现象。

在 X11 上，在窗口管理器（window manager）渲染一个窗口之前，该窗口是没有窗口边框的。窗口边框的出现，是在调用 QWidget::show()之后与窗口部件接收到第一个绘制事件（paint event）之前的某个时间点，或者根本就不会发生。

由于客户间通信协定手册（Inter-Client Communication Conventions Manual, ICCCM）未清晰描述，现存的窗口管理器对放置窗口的处理是大相径庭的。Qt 采取的策略是，发送提示给窗口管理器。而作为一个独立进程的窗口管理器，可以遵循这些提示，也可以忽略它们。X11 也没有提供一个标准的方法获取窗口渲染后的窗口边框的几何布局。Qt 的解决方案能够在目前的多数窗口管理器上工作，而如果你发现 QWidget::frameGeometry()不能正常的工作，也不要大惊小怪。

此外，X11 也没有提供最大化一个窗口的方法。Qt 的 QWidget::showMaximized()只是模拟实现了窗口的最大化。而能否真正的最大化也要依赖于 QWidget::frameGeometry()的返回结果以及窗口管理器是否能够正确地放置窗口。

应用程序可以使用 Qt 提供的几何布局函数 `QWidget::geometry()` 保存窗口部件的位置和大小, 然后在下一个会话依次调用 `QWidget::setGeometry()` 和 `QWidget::show()` 恢复显示这个窗口部件。这些函数在 Windows 系统上工作是没有问题的。然而, 在 X11 窗口系统上使用这些函数恢复显示一个窗口的大小和位置时, 并不像在 Windows 系统上工作的那样理想 (具体差异会因窗口系统的不同而变化)。这是因为, 在 X11 上, 不可见的窗口是没有窗口边框的。另一个保存、恢复窗口部件的大小和位置的做法是, 先保存函数 `QWidget::size()` 和 `QWidget::pos()` 的返回值, 然后再调用 `QWidget::resize()` 和 `QWidget::move()` 函数恢复窗口的大小和位置, 最后调用函数 `QWidget::show()` 显示窗口部件。如果还有问题的话, 只有首先调用 `QWidget::show()` 显示窗口部件, 再恢复窗口部件的大小和位置。不过这样做的后果是, 窗口先出现在一个错误的位置, 然后再闪到正确的位置。

由于管理器的不同, 相同的应用程序在使用 `QWidget` 这些几何布局函数的时候, 其行为有所差异 (不同版本的 X11 窗口系统, 窗口部件的行为也有差异)。因此上面的描述也只提供参考, 对于具体的系统平台, 应该以读者的测试结果为准。

下面, 看一个演示 `QWidget` 窗口部件类型和几何布局的例子。这个例子在 Windows 系统, 红旗 Linux 5.0 工作站和 SuSE 10.2 系统上测试通过, 但因为窗口系统的不同 (或者窗口管理器系统的版本不同), 应用程序的行为和窗口外观有所差异。

新建 KDevelop 工程 `geometry`。在设计器中绘制 ui 界面, 如图 3-31 所示。其窗口部件属性表如表 3-3 所示。

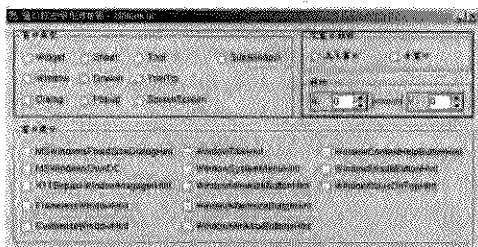


图 3-31 控制窗口 ui 界面

表 3-3 窗口部件属性表 (按从左到右、从上至下顺序, 包括部分布局管理器)

objectName	text/windowTitle	布局管理器	窗口部件类
CtrlForm	窗口标志和几何布局	垂直布局管理器	QWidget
<默认>	窗口类型	网格布局管理器	QGroupBox
widgetRadioBtn	Widget		QRadioButton
windowRadioBtn	Window		QRadioButton
dialogRadioBtn	Dialog		QRadioButton
sheetRadioBtn	Sheet		QRadioButton
drawerRadioBtn	Drawer		QRadioButton
popupRadioBtn	PopUp		QRadioButton
toolRadioBtn	Tool		QRadioButton
tooltipRadioBtn	ToolTip		QRadioButton
splashscreenRadioBtn	SplashScreen		QRadioButton
subwindowRadioBtn	SubWindow		QRadioButton



续表

objectName	text/windowTitle	布局管理器	窗口部件类
<默认>	父窗口部件	水平布局管理器	QGroupBox
nullRadioButton	无父窗口		QRadioButton
thisRadioButton	本窗口		QRadioButton
<默认>	移动	水平布局管理器	QGroupBox
<默认>	X:		QLabel
xSpinBox	——		QSpinBox
<默认>	Y:		QLabel
ySpinBox	——		QSpinBox
<默认>	窗口提示	网格布局管理器	QGroupBox
fsdialogCheckBox	MSWindowsFixedSizeDialogHint		QCheckBox
owndcCheckBox	MSWindowsOwnDC		QCheckBox
managerCheckBox	X11BypassWindowManagerHint		QCheckBox
framelessCheckBox	FramelessWindowHint		QCheckBox
customizeCheckBox	CustomizeWindowHint		QCheckBox
titleCheckBox	WindowTitleHint		QCheckBox
menuCheckBox	WindowSystemMenuHint		QCheckBox
minCheckBox	WindowMinimizeButtonHint		QCheckBox
maxCheckBox	WindowMaximizeButtonHint		QCheckBox
minmaxCheckBox	WindowMinMaxButtonsHint		QCheckBox
helpCheckBox	WindowContextHelpButtonHint		QCheckBox
shadeCheckBox	WindowShadeButtonHint		QCheckBox
topCheckBox	WindowStaysOnTopHint		QCheckBox

将 ui 文件保存为 “CtrlForm.ui”，并加入到 KDevelop 工程中。

添加控制类 CCtrlForm 的定义文件 ctrlform.h 和实现文件 ctrlform.cpp 到 KDevelop 工程中。

类 CCtrlForm 的定义文件 ctrlform.h 内容如下。

```
// chapter03/geometry/src/ctrlform.h
#ifndef _CTRLFORM_H_
#define _CTRLFORM_H_

#include <QWidget>
#include "ui_ctrlform.h"

class QPushButton;

class CCtrlForm : public QWidget,
                  public Ui_CtrlForm
{
    Q_OBJECT
public:
    CCtrlForm(QWidget* = 0);

private:
```

```

QWidget*    m_pWidget;
QPoint      m_Pos;
QSize       m_Sz;

private slots:
    void doClicked();
    void doSpinBoxChanged(int);
};
#endif

```

类 CCtrlForm 用来设置另一个窗口部件 m_pWidget 的窗口标志, 控制 m_pWidget 的显隐以及位置和大小恢复。

成员变量 m_Pos 和 m_Sz 分别保存 m_pWidget 窗口部件的位置和大小 (可以使用 QSettings 类实现永久性保存, 关于 QSettings 的使用见第 4 章的 4.6 节)。

槽函数 doClicked() 响应单选按钮和复选框的单击信号 clicked(), 完成演示窗口标志和父窗口的重置。

槽函数 doSpinBoxChanged() 控制窗口部件 m_pWidget 的移动。

下面是类 CCtrlForm 实现文件 ctrlform.cpp 的内容。

```

// chapter03/geometry/src/ctrlform.cpp.
#include <QtGui>
#include <QDebug>
#include "ctrlform.h"

CCtrlForm::CCtrlForm(QWidget* parent)
:   QWidget(parent)
{
    setupUi(this);

    m_pWidget = new QWidget(0, Qt::Widget);
    QPushButton* btn = new QPushButton(tr("关闭"));
    connect(btn, SIGNAL(clicked()), m_pWidget, SLOT(close()));
    QTextEdit* edit = new QTextEdit(tr("武汉欢迎您!"));
    QVBoxLayout* layout = new QVBoxLayout;
    layout->addWidget(edit);
    layout->addWidget(btn);
    m_pWidget->setLayout(layout);
    m_pWidget->resize(200, 100);
    m_pWidget->move(400, 500);
    m_pWidget->show();
}

```

初始化演示 QWidget 对象 m_pWidget, 设置它的大小和初始化位置, 并显示。

```

widgetRadioBtn->setChecked(true);
nullRadioBtn->setChecked(true);
xSpinBox->setRange(-100, 100);
ySpinBox->setRange(-100, 100);

```

初始化窗口部件的状态。设置“无父窗口部件”和“widget”单选按钮为选中状态; 设置 SpinBox 按钮的调节范围。



```
connect(widgetRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(windowRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(dialogRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(sheetRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(drawerRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(popupRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(toolRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(tooltipRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(splashscreenRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(subwindowRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(fsdialogCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(owndcCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(managerCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(framelessCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(customizeCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(titleCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(menuCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(minCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(maxCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(minmaxCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(helpCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(shadeCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(topCheckBox, SIGNAL(clicked()),
        this, SLOT(doClicked()));
```

关联与窗口标志有关的所有单选按钮和复选按钮的 clicked() 信号到槽函数 doFlagChanged(), 以响

应重置窗口标志的操作。

```
connect(nullptr, SIGNAL(clicked()),
        this, SLOT(doClicked()));
connect(thisRadioBtn, SIGNAL(clicked()),
        this, SLOT(doClicked()));
```

关联设置父窗口的单选按钮“无父窗口”和“本窗口”的 clicked()信号到槽函数 doClicked(), 以响应重置父窗口的操作。

```
connect(xSpinBox, SIGNAL(valueChanged(int)),
        this, SLOT(doSpinBoxChanged(int)));
connect(ySpinBox, SIGNAL(valueChanged(int)),
        this, SLOT(doSpinBoxChanged(int)));
```

关联 SpinBox 窗口部件的 QSpinBox::valueChanged()信号到槽 doSpinBoxChanged(), 以响应移动演示窗口部件的操作。

```
move(500, 400);
}
```

下面是重置演示窗口部件的窗口标志的槽函数。

```
void CCtrlForm::doFlagChanged()
{
    QPoint position = m_pWidget->pos();
    QSize sz = m_pWidget->size();
```

获取当前演示窗口部件 m_pWidget 的位置和大小, 用于再次显示窗口时恢复它们。

```
QRadioButton* btn = qobject_cast<QRadioButton*>(sender());
if(btn == nullptr)
    m_pWidget->setParent(0);
else if(btn == thisRadioBtn)
    m_pWidget->setParent(this);
```

如果选择了“无父窗口”或者“本窗口”单选按钮, 则调用函数 QWidget::setParent()重新设置演示窗口部件的父窗口部件。函数 QWidget::setParent()的调用将会隐藏演示窗口部件 m_pWidget。

```
Qt::WindowFlags flags = 0;
if(widgetRadioBtn->isChecked())
    flags = Qt::Widget;
else if(windowRadioBtn->isChecked())
    flags = Qt::Window;
else if(dialogRadioBtn->isChecked())
    flags = Qt::Dialog;
else if(sheetRadioBtn->isChecked())
    flags = Qt::Sheet;
else if(drawerRadioBtn->isChecked())
    flags = Qt::Drawer;
else if(popupRadioBtn->isChecked())
    flags = Qt::Popup;
```



```
else if(toolRadioBtn->isChecked())
    flags = Qt::Tool;
else if(tooltipRadioBtn->isChecked())
    flags = Qt::ToolTip;
else if(splashscreenRadioBtn->isChecked())
    flags = Qt::SplashScreen;
else if(subwindowRadioBtn->isChecked())
    flags = Qt::SubWindow;
```

声明一个枚举类型 `Qt::WindowFlags` 的变量 `flags`，并保存重置的窗口类型。

```
if(fsdialogCheckBox->isChecked())
    flags |= Qt::MSWindowsFixedSizeDialogHint;
if(owndcCheckBox->isChecked())
    flags |= Qt::MSWindowsOwnDC;
if(managerCheckBox->isChecked())
    flags |= Qt::X11BypassWindowManagerHint;
if(framelessCheckBox->isChecked())
    flags |= Qt::FramelessWindowHint;
if(customizeCheckBox->isChecked())
    flags |= Qt::CustomizeWindowHint;
if(titleCheckBox->isChecked())
    flags |= Qt::WindowTitleHint;
if(menuCheckBox->isChecked())
    flags |= Qt::WindowSystemMenuHint;
if(minCheckBox->isChecked())
    flags |= Qt::WindowMinimizeButtonHint;
if(maxCheckBox->isChecked())
    flags |= Qt::WindowMaximizeButtonHint;
if(minmaxCheckBox->isChecked())
    flags |= Qt::WindowMinMaxButtonsHint;
if(helpCheckBox->isChecked())
    flags |= Qt::WindowContextHelpButtonHint;
if(shadeCheckBox->isChecked())
    flags |= Qt::WindowShadeButtonHint;
if(topCheckBox->isChecked())
    flags |= Qt::WindowStaysOnTopHint;
```

保存演示窗口部件的窗口提示到枚举变量 `flags`。

```
m_pWidget->setWindowFlags(flags);
if(widgetRadioBtn->isChecked()
    && thisRadioBtn->isChecked())
{
    position = QPoint(0, 0);
}
m_pWidget->resize(sz);
m_pWidget->move(position);
m_pWidget->show();
}
```


调用函数 `QWidget::setWindowFlags()` 重新设置演示窗口部件的标志。函数 `QWidget::setWindowFlags()` 在设置窗口标志的同时也会隐藏窗口部件。

如果演示窗口部件的窗口类型为 `Qt::Widget`，并且父窗口是控制窗口，那么它将会显示在控制窗口部件的内部。为了能够将演示部件显示在可见的位置，在这种情况下，需要将演示窗口部件的位置设为(0,0)。

调用函数 `QWidget::move()` 恢复窗口部件的位置，然后调用函数 `QWidget::show()` 重新显示演示窗口部件。

从 Qt 4.2 开始，Qt 提供了保存和恢复窗口布局和状态的函数。`QWidget::saveGeometry()` 保存窗口的几何布局和最大化/全屏状态；相应的，函数 `QWidget::restoreGeometry()` 则恢复窗口的几何布局和最大化/全屏状态，该函数还会判断恢复后的几何布局是否超出了可用的屏幕布局，如果超出它会进行适当的调整。

```
void CCtrlForm::doSpinBoxChanged(int value)
{
    QSpinBox* box = qobject_cast<QSpinBox*>(sender());
    int x = m_pWidget->x();
    int y = m_pWidget->y();

    if(box == xSpinBox)
        x += value;
    else if(box == ySpinBox)
        y += value;

    m_pWidget->move(x, y);
    m_pWidget->show();
}
```

函数 `doSpinBoxChanged()` 响应移动窗口部件的操作，主要演示在不同的窗口类型、不同的窗口提示和不同的父窗口下演示窗口部件的移动情况。

修改主程序 `geometry.cpp` 文件，如下。

```
// chapter03/geometry/src/geometry.cpp.
#include <QtGui>
#include <QDebug>

#include "ctrlform.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForName("gb2312"));
    CCtrlForm form;
    form.show();
    return app.exec();
}
```

现在编译运行应用程序，测试一下窗口标志和父窗口部件的有无对用户界面的影响。



3.5 Qt 样式表

Qt 的样式表是从 Qt 4.2 开始引入的描述窗口部件外观的机制，类似于 HTML 的层叠样式表 (Cascading Style Sheets, CSS)。样式表在 Qt 的风格之上起作用 (如果使用了样式表, `QWidget::style()` 返回的 `QStyle` 为 “style sheet”), 提供了比 `QPalette` 更为灵活、更强大的机制。

样式表使用文本描述, 可以在应用程序级别和窗口部件级别设置样式表。如果在不同的级别都设置了样式表, 则 Qt 继承所有有效的样式, 这就是层叠 (cascading)。

3.5.1 样式表语法

样式表的语法和 HTML CSS 基本是一致的。Qt 的样式表对大小写不敏感, 但对类名、对象名和属性名大小写敏感。如下示例设置了所有 `QTextEdit` 对象背景是黄色的, 所有 `QPushBox` 对象文本为绿色:

```
QTextEdit { background: yellow }
QPushBox { color: green }
```

1. 样式规则

样式表包含一系列的规则, 一个样式规则由选择符和定义组成。选择符 (selector) 确定哪些窗口部件受规则影响, 定义说明了在窗口部件上应用那些属性。

例如:

```
QRadioButton {color:red}
```

在这条规则里, `QRadioButton` 是选择符, `{color:red}` 是定义。这条规则说明了 `QRadioButton` 和它的子类应该使用红色作为前景色。

几个选择符可以使用一个定义, 使用逗号分隔选择符。如:

```
QPushButton, QCheckEdit, QComboBox { color: red }
```

定义由一个或多个属性和值对组成, 中间用分号隔开, 如:

```
QPushButton { color: red; background-color: white }
```

2. 选择符类型

Qt 支持 CSS2 中所有的选择符。表 3-4 给出了常用的选择符。

表 3-4 Qt 样式表的选择符

选 择 符	示 例	说 明
通配选择符	*	匹配所有控件
类型选择符	<code>QLabel</code>	匹配 <code>QLabel</code> 及其子类
属性选择符	<code>QComboBox[editable="true"]</code>	匹配所有可以编辑的 <code>QComboBox</code> 对象
类选择符	<code>.QCheckBox</code>	只匹配 <code>QCheckBox</code> 而不匹配其子类
ID 选择符	<code>QRadioButton#red</code>	匹配对象名为 <code>red</code> 的 <code>QRadioButton</code> 对象
包含选择符	<code>QWidget QToolButton</code>	匹配所有是 <code>QWidget</code> 的子孙对象的 <code>QToolButton</code> 对象
子对象选择符	<code>QWidget > QGroupBox</code>	匹配所有是 <code>QWidget</code> 的直接子对象的 <code>QGroupBox</code> 对象

3. 子控件

对于复杂控件，可以访问它的子控件。如 QComboBox 上的下拉按钮，QSpinBox 上的向上和向下箭头。如：

```
QComboBox::drop-down { image: url(myarrow.png) }
```

上面的代码使用了自定义的下拉按钮图像。::是 CSS3 中的伪元素。

4. 伪状态

选择符可以包含伪状态来表示窗口部件的状态。伪状态在选择符之后，以冒号分隔，下面定义了当鼠标在 QPushButton 上悬停时的规则：

```
QPushButton:hover { color: white; }
```

5. 冲突解决

当不同的规则应用到相同的属性时，样式表就产生了冲突。在这种情况下，特定的规则比通用的规则优先；伪状态比没有伪状态的优先；如果级别相同，则最后一个规则优先。冲突解决按照 CSS2 规范进行。

6. 层叠

样式表可以在 QApplication 级别设置，也可以在父窗口部件，子窗口部件级别设置。实际应用样式时，则合并这几个级别的样式。当有冲突时，窗口部件自身的样式优先使用，接下来是父窗口部件，祖先窗口部件，依此类推。

7. 盒子模型

窗口部件和子窗口部件支持背景 (background)、边框 (border)、边距 (margin)、填衬 (padding)，图 3-32 显示了样式表的盒子模型。

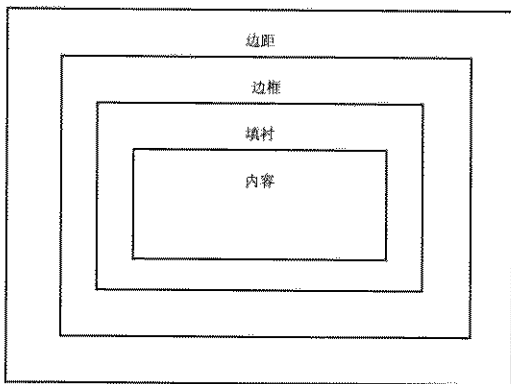


图 3-32 Qt 样式表的盒子模型

从图 3-32 可以看到边距、边框、填衬和内容之间的关系。默认情况下，margin, border-width, padding 属性为 0，也就是说这几个矩形框是重合的。如果指定了窗口部件的背景，默认情况下背景只对边框



内部区域起作用，但可以通过改变 background-clip 属性来指定不同的区域填充。

3.5.2 样式表的应用

应用样式表很简单，只需要调用 QApplication 或 QWidget 的 setStyleSheet() 函数将样式表设置到应用程序或窗口部件就可以了，工作量较多的是设计样式表。下文将给出使用样式表的实际例子。

为了使用样式表，首先用 Qt 设计器设计一个用户界面。要实现的应用程序是一个交规模拟考试程序，当然这个程序并不能完成真正的考试，而只有一个用户界面来演示样式表的使用。设计的界面如图 3-33 所示。



图 3-33 应用样式表的用户界面

使用 Qt 设计器设计界面后，窗口部件的定义代码就非常简洁，代码如下所示：

```
// chapter03/css/testwidget.h
#ifndef TESTWIDGET_H_
#define TESTWIDGET_H_

#include <QWidget>
#include "ui_test.h"

class TestWidget : public QWidget
{
    Q_OBJECT

public:
    TestWidget(QWidget * parent = 0, Qt::WindowFlags f = 0);

protected slots:
    void timerEvent(QTimerEvent *event);

private slots:
    void on_comboBox_currentIndexChanged(const QString & text);
```

```
private:
    Ui::test ui;
    short remain;
};
#endif /*TESTWIDGET_H_*/
```

类中的 `timerEvent()` 用来显示考试还剩下的时间。用户选择组合框中的样式后，应用程序将应用新的样式表。类实现如下：

```
// chapter03/css/testwidget.cpp
#include <QtGui>
#include "TestWidget.h"

TestWidget::TestWidget(QWidget * parent, Qt::WindowFlags f)
: QWidget(parent, f)
{
    ui.setupUi(this);
    remain = 45;
    startTimer(50*1000);
}

// 改变样式表
void TestWidget::on_comboBox_currentIndexChanged(const QString & text)
{
    QFile file(":/qss/" + text.toLower() + ".qss");
    file.open(QFile::ReadOnly);
    QString styleSheet = tr(file.readAll());
    QApplication::setStyleSheet(styleSheet);
}

void TestWidget::timerEvent(QTimerEvent *event)
{
    --remain;
    ui.lblRemain->setText(QString(tr("还剩%1 分钟")).arg(remain));
}
```

应用程序中根据用户从 `QComboBox` 选择的样式表而设置应用程序的样式表。这里把样式表作为资源文件。实际上，样式表也可以作为独立文件来使用。

可以看出，应用样式表的代码编写非常简单，主要的工作还是在设计样式表本身。这里给出其中一个样式表的文本：

```
/*设置 QFrame 的背景色*/
QFrame {
    background-color: khaki;
}

/*设置 QPushButton 为圆角矩形，并具有边框*/
QPushButton {
    background-color: yellowgreen;
    border-width: 2px;
    border-color: seagreen;
```



```
border-style: solid;
border-radius: 5;
padding: 3px;
min-width: 9ex;
min-height: 2.5ex;
}
/*当鼠标在 QPushButton 上悬停时, 按钮改变颜色*/
QPushButton:hover {
    background-color: springgreen;
}
/*当 QPushButton 按下时, 显示按下的效果*/
QPushButton:pressed {
    padding-left: 5px;
    padding-top: 5px;
    background-color: palegreen;
}
/*设置 QLabel 的字体和前景色、背景色*/
QLabel {
    font:bold;
    font-family: 隶书;
    font-size: 16px;
    color: red;
    background-color: palevioletred;
}
/*设置 QComboBox 和 QSpinBox 的背景色和选中时的颜色*/
QComboBox, QSpinBox {
    background-color: plum; /* 暗紫色 */
    selection-color: mediumvioletred;
    selection-background-color: lightcoral;
}
/*在 QRadioButton 选中时, 改变颜色*/
QRadioButton:checked {
    color:deepskyblue;
}
/*当鼠标在 QRadioButton 上悬停时, 改变颜色*/
QRadioButton:hover {
    color:white;
    background-color: limegreen;
}
/*设置 QGroupBox 的边框线型 */
QGroupBox {
    border-style:dashed;
}
```

上面的颜色名, 如 `limegreen`, 都是 CSS 中定义的标准颜色名。如果要表示非标准的颜色, 可以使用十六进制数表示。要设置一个好的样式, 除了要对 Qt 的样式表语法熟悉以外, 还要具有一定的美学知识, 这已经超出了本书的讨论范围。

3.6 Qt 对象模型

标准 C++ 对象模型 (C++ object model) 非常高效地支持对象范式 (object paradigm)。然而,在某些方面却表现不足,比如信号的传递、事件的传递和处理等。GUI 编程既要求运行时的高效性,又要有更好的灵活性。Qt GUI 编程语言结合了 Qt 对象模型 (Qt Object Model) 的灵活性以及标准 C++ 运行时的高效性。

为了更好地满足 GUI 图形用户界面编程,Qt 的对象模型在标准 C++ 的基础上新增了一些特性:

- 元对象系统 (Meta-Object System), 提供
 - 对象通信机制: 信号和槽;
 - 动态的对象转换 (dynamic cast)。
- 可查询和可设计的对象属性 (object properties);
- 层次结构、可查询的对象树 (object trees);
- 安全的指针 (QPointer), 在对象销毁时, 该指针自动设置为 NULL 值, 这有别于 C++ 的指针;
- 强大的事件和事件过滤器;
- 支持国际化的文本转换机制;
- 定时器机制, 使得多个任务 (tasks) 可以集成在一个事件驱动的 (event-driven) GUI 中。

Qt 对象模型中的大多数特性都是通过标准 C++ 技术实施的。其他的特性, 像对象通信机制、动态属性系统 (dynamic property system) 等需要 Qt 自身提供的元对象编译器 (Meta-Object Compiler, moc) 的支持。

本小节主要介绍元对象系统、对象属性系统和对象树, 其他内容将在相关章节中陆续阐述。

3.6.1 元对象系统

Qt 元对象系统提供了对象间的通信机制 (信号和槽)、运行时类型信息和动态属性系统的支持, 是标准 C++ 的一个扩展, 它使得 Qt 更好地实现 GUI 图形用户界面编程。Qt 的元对象系统不支持 C++ 模板, 尽管模板扩展了标准 C++ 的功能, 但是元对象系统提供了模板无法提供的一些特性。

Qt 的元对象系统基于三个事实:

- 基类 QObject, 任何想使用元对象系统功能的类必须继承自 QObject;
- Q_OBJECT 宏, Q_OBJECT 宏必须出现在类的私有声明区, 以启动元对象的特性;
- 元对象编译器 (Meta-Object Compiler, moc), 为 QObject 子类实现元对象特性提供必要的代码实现。

下面以 CFindFileForm 类的头文件 findfileform.h 为例, 看一下 Qt moc 工具的工作过程:

01 在编译应用程序的时候, 由 make 工具调用 moc 工具进行处理;

02 moc 工具读取头文件 findfileform.h, 看是否包含 Q_OBJECT 宏, 如果包含则进行下一步处理, 否则 moc 放弃对 findfileform.h 的处理;

03 moc 根据 findfileform.h 生成另一个 C++ 源文件 (默认的情况下名字命名为 moc_findfileform.cpp), 该源文件包含了元对象代码的实现;

04 接着, C++ 编译器处理 moc_findfileform.cpp 文件, 生成中间文件 moc_findfileform.o;

05 最后, 连接器将 moc_findfileform.o 同其他应用程序的中间文件连接起来, 生成可执行应用程序文件。



元对象系统除了支持信号/槽机制和对象的动态转换（这两个内容已经在前面的章节中讲到，不再赘述）外，还提供一些其他的特性，包括：

- `QObject::metaObject()`，返回一个类的元对象；
- `QMetaObject::className()`，在运行时以字符串的形式返回类名，但不需要 C++ 编译器的 RTTI 的支持（RTTI 会降低 C++ 应用程序的运行效率）；
- `QObject::inherits()`，判断一个对象是否是某个指定类的子类实例；
- `QObject::tr()` 和 `QObject::trUtf8()`，进行国际化的字符串翻译，等等。

下面，看一下使用元对象特性的一个例子。

```
// chapter03/metaobj/src/metaobj.cpp
#include <QDebug>
#include <QtGui>
#include <QtCore>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForName("gb18030"));

    QObject* obj = new QLabel;
    const QMetaObject* mo = obj->metaObject();
    qDebug() << QObject::tr("类名: %1").arg(mo->className());
    qDebug() << QObject::tr("是否继承自 QWidget: %1")
        .arg(obj->inherits("QWidget") ? QObject::tr("是") : QObject::tr("否"));
    return 0;
}
```

程序运行的结果为：

```
"类名: QLabel"
"是否继承自 QWidget: 是"
```

3.6.2 属性系统

Qt 的属性系统建立在 Qt 元对象系统的基础上，可以工作在 Qt 支持的任何平台，并且能够使用任何标准的 C++ 编译器。

在类的定义文件中，通过使用 `Q_PROPERTY()` 宏声明一个属性，而且只有继承自 `QObject` 的子类才可以使用 Qt 的属性系统。一个属性类似于类的数据成员，不过同数据成员相比，属性具有下列一些特征：

- 必须有一个读（read）函数；
- 一个可选的写（write）函数（只读属性没有写函数）；
- 一个可选的重置（reset）函数，将属性恢复到一个默认值，该函数必须返回 void 并且没有任何参数；
- 一个可选的“DESIGNABLE”特性，表明该属性是否在 GUI 构造器（例如，Qt 设计器）中可用的，可写属性的默认值为 true，只读属性的默认值为 false；
- 一个可选的“SCRIPTABLE”特性，表明脚本引擎（scripting engine）是否可以访问该属性，可

写属性的默认值为 `true`，只读属性默认值为 `false`；

- 一个可选的“STORED”特性，表明该属性是否持久的，即当存储一个对象的状态时，是否保存该属性的值，该特性仅仅对可写属性有意义，默认值为 `true`。

读、写和重置函数可以像一般的成员函数一样，既可以是虚的，也可以从父类继承。不同的是，在多继承情况下，读、写和重置函数必须继承自第一个父类（即类的定义中，继承类列表中最左边的类）。

下面，看一个使用对象属性的例子（KDevelop 工程名：`property`）。首先看一下类的定义文件 `weapon.h`。

```
// chapter03/property/src/weapon.h
#include <QObject>

#ifndef _WEAPON_H_
#define _WEAPON_H_

typedef enum {
    nil,
    ready,
    fired,
    exceptional
} Status;

class CWeapon : public QObject
{
    Q_OBJECT
    Q_PROPERTY(Status status READ getStatus WRITE setStatus)
    Q_ENUMS(Status)

public:
    CWeapon(QObject *parent = 0);
    void setStatus(Status s);
    Status getStatus() const;

private:
    Status status;
};
#endif
```

头文件中定义了一个 `Status` 枚举类型，表示武器的状态：无弹、就绪、开火和异常。该枚举类型之所以定义在类 `CWeapon` 的外面，是为了方便在 `CWeapon` 作用域之外使用它（否则，`getStatus()` 函数的返回类型必须是 `CWeapon::Status`）。

宏 `Q_PROPERTY()` 向元对象系统注册一个属性，它的语法如下，

```
Q_PROPERTY(Type name
            READ getFunction
            [WRITE setFunction]
            [RESET resetFunction]
            [DESIGNABLE bool])
```

```
[SCRIPTABLE bool]
[STORED bool()]
```

其中，类型 `Type` 必须是 `QVariant` 支持的类型或者是同属性所在类一起声明的枚举类型。而且，枚举类型必须通过 `Q_ENUM()` 宏在属性系统进行注册。

另一个类似于 `Q_ENUMS()` 的宏是 `Q_FLAGS()`。与 `Q_ENUMS()` 不同的是，`Q_FLAGS()` 除了向属性系统注册枚举类型外，它还会将这个枚举标识为一组“标记 (flags)”，这些枚举值可以进行位操作。

宏 `Q_PROPERTY()` 只是向元对象系统注册一个属性，因此还需要对属性及其操作进行声明。此外，`Q_PROPERTY()` 宏中的属性名字 `name` 可以和私有成员的名字不同，而操作名字必须相同。例如，

```
Q_PROPERTY(Status status READ getStatus WRITE setStatus)
```

而私有区的成员可以声明为：

```
private:
    Status m_Status;
    Status m_St;
```

不过，读写函数的实现必须做出相应的修改，对属性的读、写等操作必须针对 `m_Status`（如果 `m_Status` 是属性 `status` 对应的成员的话）。而读、写函数名字必须是 `getStatus` 和 `setStatus`。对类 `CWeapon` 的外部而言，只有名字 `status` 是可见的（比如在 Qt 设计器或者在脚本当中，可以直接对 `CWeapon` 的 `status` 属性进行修改。具体详见第 18 章“Qt 插件”和第 19 章“脚本——QtScript”的相关内容）。

下面是类 `CWeapon` 的实现文件 `weapon.cpp` 的内容。

```
// chapter03/property/src/weapon.cpp.
#include "weapon.h"
```

```
CWeapon::CWeapon(QObject * parent)
:   QObject(parent)
{
}
```

构造函数完成对象的初始化。

```
Status CWeapon::getStatus() const
{
    return status;
}
```

属性的读函数获取属性的值。

```
void CWeapon::setStatus(Status s)
{
    status = s;
}
```

属性的写函数设置属性的值。

除了使用 `Q_PROPERTY()` 宏定义一个属性外，还可以通过 `QObject::setProperty()` 函数在运行时添加一个动态属性。相应的，`QObject::property()` 可以获取动态属性的值。

下面，看一下对象动态属性的例子。

```
// chapter03/property/src/property.cpp.
#include <QtCore>
#include "weapon.h"

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    CWeapon weapon;
    weapon.setStatus(fired);
    qDebug() << "status " << weapon.getStatus();
    weapon.setProperty("own", 1);
    qDebug() << "own" << weapon.property("own").toInt();

    return 0;
}
```

上面的程序将会给 CWeapon 对象 weapon 添加一个名字为“own”的动态属性，并且输出该动态属性的值。

编译运行应用程序，输出结果为：

```
status 2
own 1
```

3.6.3 对象树

Qt 提供了一种机制，能够自动、有效地组织和管理继承自 QObject 的 Qt 对象（包括继承自 QObject 类的自定义类），这种机制就是 Qt 对象树。当应用程序生成一个具有父窗口部件（严格地讲应该是父 QObject 对象，不一定是 GUI 对象，在此统称父窗口部件。子窗口部件类推）的 Qt 对象的时候（通过在构造函数中的 QObject* parent 参数指定），这个新的对象将会被添加到父窗口部件的孩子列表中（通过 QObject::children() 能够获取对象的子窗口部件），而当对象的父窗口部件被销毁的时候，Qt 的对象树机制能够保证也销毁它所有的子窗口部件。Qt 的对象树机制对于图形用户界面编程是非常有用的，例如，一个 QShortcut 对象被设置为一个窗口部件的孩子，当用户关闭并销毁该窗口部件时，这个 QShortcut 对象也就被销毁了，这极大地减少了程序员的工作，能够将主要精力放到系统的业务上，提高了编程效率，也增强了系统的稳健性。

C++ 编程一个重要的方面是防止内存泄露，方法是通过 new 操作创建的对象必须通过 delete 操作进行销毁。尽管程序员已经在这方面做的很小心，但好像做得并不出色。而 Qt 的对象树机制可以减轻内存泄露带来的压力。在进行 Qt GUI 编程的时候，可以选择 Qt 对象的销毁方式：

- 销毁 Qt 对象树中的顶层 Qt 对象，由对象树机制保证子窗口部件的销毁；
- 在代码中显示销毁（delete）子窗口部件。

对于显式销毁子窗口部件对象，要保证先销毁子窗口部件之后，再销毁它的父窗口部件。否则的话，将会导致语义未定义。这是因为销毁一个不存在的窗口部件造成的（因为先销毁父窗口部件的话，子窗口部件也被销毁了，当再显式销毁子窗口部件的时候，相当于销毁一个不存在的对象，语义是未定义的）。

下面，看一下利用对象树销毁窗口部件的例子。



```
// chapter03/objtree/exam1.  
#include <QtGui>  
#include <QDebug>  
  
int main(int argc, char *argv[])  
{  
    QApplication app(argc, argv);  
  
    QDialog* dlg = new QDialog(0);  
    qDebug() << "dlg(1) = " << dlg;
```

创建一个对话框堆窗口对象（父设置为 0），输出该对话框对象。输出结果为：

```
dlg(1) = QDialog(0x8bca500)
```

注意，括号中的地址会因环境的差异而会不同。

```
QTableWidget* tbl = new QTableWidget(dlg);  
qDebug() << "tbl(1) = " << tbl;
```

创建一个 QTableWidget 堆对象，父设置为上面创建的对话框窗口，然后输出该表格窗口部件。输出结果为：

```
tbl(1) = QTableWidget(0x8bcd080)
```

```
dlg->exec();
```

执行对话框（在出现的对话框中执行关闭操作）。

```
qDebug() << "dlg(2) = " << dlg;  
qDebug() << "tbl(2) = " << tbl;
```

再次输出对话框窗口和表格窗口部件，输出结果分别同 dlg(1)和 tbl(1)的结果相同。说明关闭一个窗口，只会隐藏该窗口，而不会销毁。

```
dlg->exec();  
delete dlg;
```

再次执行对话框，然后销毁对话框窗口 dlg。

```
Q_ASSERT(dlg != NULL);  
Q_ASSERT(tbl != NULL);
```

尽管窗口部件被销毁了，但对象指针没有被赋值为 0，这是由编译器决定的。注意，C++标准约定，允许销毁一个值为 NULL 的指针。

```
qDebug() << "after delete parent";  
delete tbl;
```

销毁表格子窗口部件 tbl，结果会出现段错误。说明对象树在起作用，已经将表格子窗口部件销毁掉了。输出结果为：

```
after delete parent  
段错误
```

```
    return 0;
}
```

现在，改变一下销毁对象的顺序：先销毁子窗口部件，再销毁父窗口部件。

```
// chapter03/objtree/exam2.
#include <QtGui>
#include <QDebug>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QDialog* dlg = new QDialog(0);
    qDebug() << "dlg(1) = " << dlg;

    QTableWidgetItem* tbl = new QTableWidgetItem(dlg);
    qDebug() << "tbl(1) = " << tbl;

    dlg->exec();
    qDebug() << "dlg(2) = " << dlg;
    qDebug() << "tbl(2) = " << tbl;

    delete tbl;
    dlg->exec();
```

销毁表格子窗口部件 `tbl`，再次执行对话框。这时，呈现的对话框没有表格了。说明表格子窗口部件已经被销毁掉，而父窗口仍然可以运行。

```
Q_ASSERT(dlg != NULL);
Q_ASSERT(tbl != NULL);
qDebug() << "after delete child";
delete dlg;
```

销毁父窗口 `dlg`，应用程序运行正常。

```
    return 0;
}
```

也可以隐含设置 Qt GUI 对象间的父子关系，该方法不需要在构造函数中设置参数 `<QWidget* parent>`。

```
// chapter03/objtree/exam3.
#include <QtGui>
#include <QDebug>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QDialog* dlg = new QDialog(0);
    QTableWidgetItem* tbl = new QTableWidgetItem(dlg);
```



```
QHBoxLayout* layout = new QHBoxLayout;
layout->addWidget(tbl);
dlg->setLayout(layout);

QList<QObject*> list = dlg->children();
QDebug() << "dialog's children: ";
for(int i=0; i<list.size(); ++i)
    qDebug() << list.at(i);
```

函数 `QObject::children()` 获取 Qt 对象的子窗口部件。

输出结果为：

```
dialog's children:
QWidget(0x83a8f50)
QHBoxLayout(0x84705f0)

list = layout->children();
qDebug() << "layout's children: ";
for(int i=0; i<list.size(); ++i)
    qDebug() << list.at(i);
```

输出 `QHBoxLayout` 对象的子窗口部件，输出结果为：

```
layout's children:
```

即 `QHBoxLayout` 对象 `layout` 的子窗口部件为空。说明通过 `QHBoxLayout::addWidget()` 添加的窗口部件并不是由 `QHBoxLayout` 的对象树进行组织管理。

```
delete tbl;
list = dlg->children();
QDebug() << "afterdelete tablewidget, dialog's children: ";
for(int i=0; i<list.size(); ++i)
    qDebug() << list.at(i);
```

销毁表格子窗口部件后，输出对话框的子窗口部件。输出结果为

```
afterdelete tablewidget, dialog's children:
QHBoxLayout(0x84705f0)

return 0;
}
```

注意，Qt GUI 中“父窗口部件/子窗口部件”关系是区别于面向对象编程中的对象继承关系“父对象和子对象”的。

3.7 小 结

`QWidget` 类是其他 Qt GUI 类的基类，本章介绍了它的类型以及窗口部件的几何布局。在这一章，还学习了在 Qt 设计器中绘制一个自定义的窗口部件，并引入到应用程序中。Qt 的样式表继 Qt 在界面中使用 HTML 标签以来又一次结合了 Web 设计的一些优点。最后，详细解释了 Qt 的对象模型，阐述了元对象系统、属性系统和对象树的基本内容。这一章的内容对了解 Qt 的基本原理具有很大的帮助。

第 4 章 程序主窗口——QMainWindow

在前面几章中,学习了 Qt 的一些基础知识。在这一章,将学习使用 Qt 的主窗口部件 QMainWindow 进行 GUI 应用程序开发。通过一个简单文本编辑器的例子,学习使用 GUI 用户界面主窗口的菜单、工具栏、状态栏等。

4.1 QMainWindow 主窗口框架

Qt 的 QMainWindow 类提供了一个应用程序主窗口,包括一个菜单栏(menu bar)、多个工具栏(tool bars)、多个锚接部件(dock widgets)、一个状态栏(status bar)以及一个中心部件(central widget),其界面布局如图 4-1 所示。

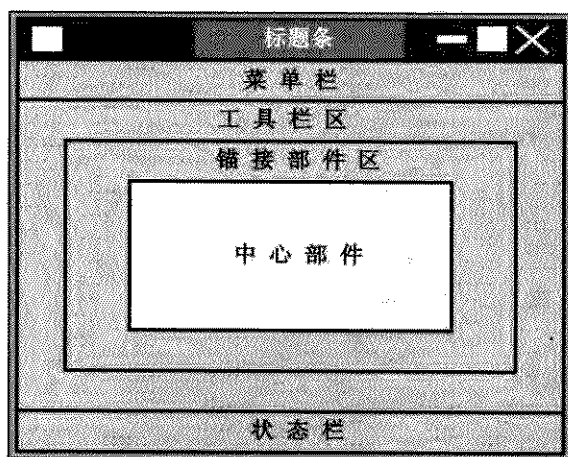


图 4-1 Qt 主窗口框架布局

1. 菜单栏

菜单是一系列命令的列表。为了实现菜单、工具栏按钮、键盘快捷方式等命令的一致性,Qt 使用动作(Action)来表示这些命令。Qt 的菜单就是由一系列的 QAction 动作对象构成的列表。而菜单栏则是包容菜单的面板,它位于主窗口的顶部、主窗口标题栏的下面。一个主窗口最多只有一个菜单栏。

2. 状态栏

状态栏通常显示 GUI 应用程序的一些状态信息,它位于主窗口的最底部。可以在状态栏上添加、使用 Qt 窗口部件。一个主窗口最多只有一个状态栏。

3. 工具栏

工具栏是由一系列的类似于按钮的动作排列而成的面板，它通常有一些经常使用的命令（动作）组成。工具栏的位置处在菜单栏的下面，状态栏的上面。工具栏可以停靠在主窗口的左、右、上、下四个方向上。一个主窗口可以包含多个工具栏。

4. 锚接部件

锚接部件作为一个容器来使用，以包容其他窗口部件来实现某些功能。比如，Qt 设计器的属性编辑器、对象监视器等都是由锚接部件包容其他的 Qt 窗口部件来实现的。它处在工具栏的内部，可以作为一个窗口自由地浮动在主窗口的上面，也可以像工具栏一样停靠在主窗口的左、右、上、下四个方向上。一个主窗口可以包含多个锚接部件。

5. 中心部件 (Central Widget)

中心部件处在锚接部件的内部、主窗口的中心。一个主窗口只有一个中心部件。

注意，主窗口 `QMainWindow` 具有自己的布局管理器，因此在 `QMainWindow` 窗口上设置布局管理器或者创建一个父窗口部件为 `QMainWindow` 的布局管理器都是不允许的。但可以在主窗口的中心部件上设置布局管理器。

为了控制主窗口工具栏和锚接部件的显现，在默认情况下，`QMainWindow` 主窗口提供了一个上下文菜单 (Context Menu)。通常，通过在工具栏或锚接部件上单击鼠标右键就可以激活该上下文菜单；也可以通过函数 `CMainWindow::createPopupMenu()` 来激活该菜单。此外，还可以重写 `CMainWindow::createPopupMenu()` 函数，实现自定义的上下文菜单。

4.2 Qt 设计器绘制主窗口

创建应用程序主窗口界面主要有两种方式：

(1) 全部代码生成。单继承 `QMainWindow` 类，在子类的实现文件中使用代码创建应用程序主窗口的菜单、工具栏、锚接部件以及状态栏等并设置它们的属性；使用单继承 Qt 窗口部件类的方法生成中心部件并添加到主窗口中。

(2) Qt 设计器绘制应用程序主窗口。在 Qt 设计器中添加菜单（以及子菜单和动作）、工具栏（以及动作）、锚接部件（以及子窗口部件）、状态栏（目前，Qt 设计器没有提供状态栏的设计编辑功能，比如无法将窗口部件直接拖放到主窗口的状态栏上）等并设置它们的属性，以及关联一些基本的信号和槽；然后采用前面介绍的“单一继承方式”或“多继承方式”实现应用程序主窗口的代码。

一般的，第二种方法创建应用程序主窗口是比较快捷的。下面在 Qt 设计器中对应用程序主窗口进行初步的设计和绘制。

在 Qt 设计器中的“new form”对话框中选择“Main Window”选项，单击“create”按钮，创建应用程序的主窗口。选择“Tools”菜单的“Object Inspector”子菜单，在 Qt 设计器的对象监视器中，可以看到，应用程序主窗口的对象名字为 `MainWindow`，同时 Qt 设计器也已经创建了一个中心部件 `centralwidget`、一个菜单栏 `menubar` 和一个状态栏 `statusbar`。通过 Qt 设计器的属性编辑器，可以修改这些窗口部件对象的属性（包括它们的名字等）。在此，采用默认值。

将创建的主窗口 `ui` 文件保存在下面创建的 KDevelop 工程的 `src` 目录下（`chapter04/designmainwindow/src`），文件名字为“`mainwindow.ui`”。

现在, 将主窗口引入到应用程序中, 以便一边在 Qt 设计器中绘制主窗口界面, 一边可以运行应用程序直接查看绘制效果(在 Qt 设计器中也可以预览绘制的 GUI 用户界面, 通过菜单“Form”“Preview”可以做到, 或者直接在键盘上激活快捷键“Ctrl + R”)。

01 建立 KDevelop 工程 designmainwindow:

02 新建 CMainWindow 类的头文件 mainwindow.h, 并添加到工程中:

```
// chapter04/designmainwindow/src/mainwindow.h
#ifndef _MAINWINDOW_H_
#define _MAINWINDOW_H_

#include "ui_mainwindow.h"

class CMainWindow : public QMainWindow,
                   public Ui_MainWindow
{
    Q_OBJECT
public:
    CMainWindow(QWidget* = 0);
};
#endif
```

03 新建 CMainWindow 类的实现文件 mainwindow.cpp, 并添加到工程中。

```
// chapter04/designmainwindow/src/mainwindow.cpp
#include <QtGui>
#include "mainwindow.h"

CMainWindow::CMainWindow(QWidget* parent)
    : QMainWindow(parent)
{
    setupUi(this);

    showMaximized();
}
```

QMainWindow::showMaximized()函数最大化显示应用程序主窗口界面。

04 在 mainwindow 工程中修改 src 目录下的主程序源文件 designmainwindow.cpp:

```
// chapter04/designmainwindow/src/designmainwindow.cpp
#include <QtGui>
#include <QtCore>

#include "mainwindow.h"
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForName("gb18030"));
    QTranslator translator;
}
```



```
QStringList environment = QProcess::systemEnvironment();
QString str;
bool bFinded = false;
foreach(str, environment)
{
    if(str.startsWith("QTDIR="))
    {
        bFinded = true;
        break;
    }
}
if(bFinded)
{
    str = str.mid(6);
    bFinded = translator.load("qt_" + QLocale::system().name(),
                              str.append("/translations/"));
    if(bFinded)
        qApp->installTranslator(&translator);
    else
        qDebug() << QObject::tr("没有支持中文的 Qt 国际化翻译文件!");
}
else
{
    qDebug() << QObject::tr("必须设置 QTDIR 环境变量!");
    exit(1);
}

CMainWindow mainWindow;

return app.exec();
}
```

05 将 `mainwindow.ui` 文件加入到 `qmake` 工程中。

现在编译运行应用程序，可以看到一个没有任何内容的主窗口。

4.2.1 菜单

在一个大型的应用程序中，菜单往往是不可或缺的，它为客户提供了一简便快捷的操作模式。菜单有两种：主菜单（或下拉菜单）和上下文菜单。主菜单的位置是固定的，即在应用程序主界面的顶部；上下文菜单一般在用户单击鼠标右键时出现在鼠标的位置，应用更加灵活方便。

下面学习创建应用程序主窗口的主菜单。

默认情况下，Qt 设计器已经生成一个名字为“menubar”的主菜单栏，如图 4-2 所示。

通过在菜单栏上单击鼠标右键，可以选择 Qt 设计器的上

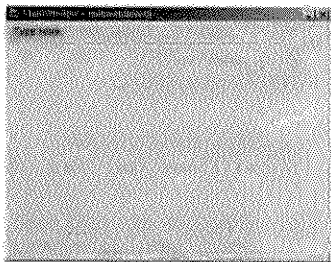


图 4-2 默认的主菜单

下文菜单命令“Remove Menu Bar”移除菜单（如图 4-3 所示）；移除之后也可以通过上下文菜单命令“Create Menu Bar”再次添加一个新的菜单条（如图 4-4 所示）。

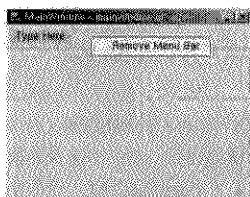


图 4-3 移除主窗口菜单条

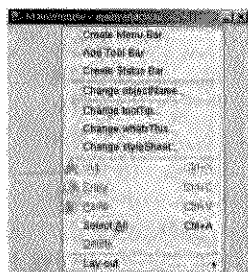


图 4-4 添加主窗口菜单条

接下来要做的是，在主菜单栏上添加/编辑菜单和子菜单。

下面，按步骤建立“文件”菜单。

01 在主窗口上双击“type here”，在出现的文本框中输入第一个菜单的名字“文件(&F)”，并按回车键。括号中的“&F”表示将字符 F 作为“文件”菜单显示文本的助记符（在字母 F 下加下划线，即显示为“文件(F)”，作用是将“Alt + F”设置为加速键（accelerator）。当应用程序处在活动状态时，通过按下加速键“Alt + F”，就可以将“文件”菜单激活；或者主窗口的菜单栏处在活动的状态下（通过按“Alt”键就可以将菜单栏激活），在键盘上直接按下“F”键就可以激活“文件”菜单。

02 这时 Qt 设计器会自动显示需要建立的“文件”菜单。接着，输入菜单命令的名字“新建(&N)”，“打开(&O)...”和“关闭(&C)”。“&N”，“&O”和“&C”定义了子菜单的加速键，它们必须在“文件”菜单处在激活的状态下才有效。即在文件菜单活动的状态下，通过直接按键“N”、“O”和“C”可以激活“新建”、“打开”和“关闭”菜单命令。

03 双击菜单底部的“add separator”，在“关闭”菜单命令后面添加一条分隔线。

04 建立“保存(&S)...”和“另存为(&A)...”菜单命令。

05 加一条分隔线，建立“退出(&X)”菜单命令。

06 在对象监视器（Object Inspector）中单击需要修改属性的动作 QAction（对应于菜单命令），接着在“属性编辑器”中修改菜单动作的属性；在此，修改动作的对象名字（objectName），并添加必要的快捷键（shortcut 属性，选中 shortcut 属性后，直接按键就可以了，比如“Ctrl+S”，如图 4-5 所示）。文件菜单属性表如表 4-1 所示。

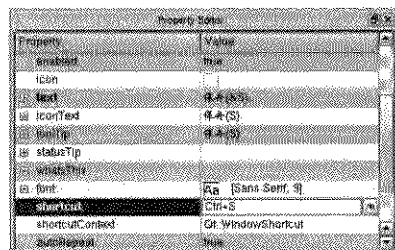


图 4-5 设置快捷键

表 4-1 “文件”菜单属性表

菜单显示文本(text)	对象名字 (objectName)	快捷键 (shortcut)	Qt 类
文件(&F)	menu_F	—	QMenu
新建(&N)	actNew	Ctrl+N	QAction
打开(&O)	actOpen	Ctrl+O	QAction
关闭(&C)	actClose	—	QAction
保存(&S)	actSave	Ctrl+S	QAction
另存为(&A)	actASave	—	QAction
退出(&X)	actQuit	—	QAction

笔者建议一边添加菜单命令，一边修改动作的对象名字，以免造成混乱，防止对象名字的张冠李戴。如果实在不知道那个菜单命令对应哪个动作，可以在相应的菜单命令上单击鼠标右键，这时会出现一个文本为“Remove action ‘xxx’”的上下文菜单命令，这样就知道单击的这个菜单动作的对象名称叫 xxx 了。如果仅仅修改动作的显示文本、对象名称或者添加图标，也可以在动作编辑器中双击某个动作，然后在“Edit Action”对话框中进行编辑，如图 4-6 所示。

建好后的菜单如图 4-7 所示。



图 4-6 编辑动作

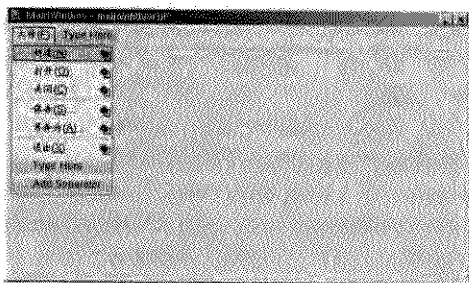


图 4-7 创建“文件”菜单后的主窗口

现在通过相同的方式，添加“编辑”菜单，如图 4-8 所示。“编辑”菜单属性如表 4-2 所示。添加“工具”菜单，如图 4-9 所示。“工具”菜单属性如表 4-3 所示。

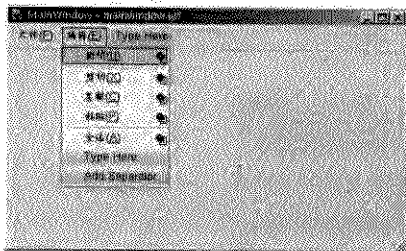


图 4-8 创建“编辑”菜单后的主窗口

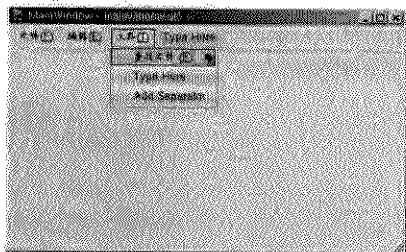


图 4-9 创建“工具”菜单后的主窗口

表 4-2 “编辑”菜单属性表

菜单显示文本(text)	对象名字 (objectName)	快捷键 (shortcut)	Qt 类
编辑(&E)	menu_E		QMenu
撤销(&U)	actUndo	Ctrl+Z	QAction
剪切(&T)	actCut	Ctrl+X	QAction
复制(&C)	actCopy	Ctrl+C	QAction
粘贴(&P)	actPaste	Ctrl+V	QAction
全选(&A)	actAll	Ctrl+A	QAction

表 4-3 “工具”菜单属性表

菜单显示文本(text)	对象名字 (objectName)	快捷键 (shortcut)	Qt 类
工具(&T)	menu_T		QMenu
查找文件(&F)	actFind	Ctrl+F	QAction

在上述过程中,如果要删除一个菜单(或菜单命令),可以在相应的菜单(或菜单命令)上单击右键,在弹出的快捷菜单中选择“Remove action ‘xxx’”或者“Remove Menu ‘xxx’”就可以了。

保存绘制的主窗口界面文件,编译运行应用程序,看一下刚刚绘制的应用程序主窗口菜单。

4.2.2 工具栏

对于用户而言,工具栏提供了实现用户操作意图的一种更简捷的操作模式,不用拉下菜单或者记住具体的按键,工具栏就出现在用户的面前。为了适应不同的用户,大型的应用程序往往提供很多的工具栏,而工具栏的缺点就是它要占据 GUI 用户界面空间。因此,往往只有用户最常用的命令放置在工具栏中,同时工具栏也可以根据用户的喜好关闭或打开。

现在,在 Qt 设计器中建立两个工具栏:“文件”工具栏和“编辑”工具栏,分别对应于应用程序主窗口的“文件”菜单和“编辑”菜单的某些常用功能。

首先添加“文件”工具栏。

01 Qt 设计器中,在应用程序主窗口中单击鼠标右键,在弹出的上下文菜单中选择“Add Tool Bar”命令,在应用程序主窗口的顶部、主菜单的下面添加一个空白的工具栏,在属性编辑器中修改工具栏的对象名字为“fileToolBar”;

02 接下来,设计工具栏的动作(actions)。因为在绘制主菜单的时候,Qt 设计器已经生成了“文件”菜单的动作,因此我们可以直接利用已存在的动作。选择 Qt 设计器的菜单“Tools”|“Action Editor”,打开动作编辑器。可以看到所有的 Qt 设计中创建的动作,如图 4-10 所示。

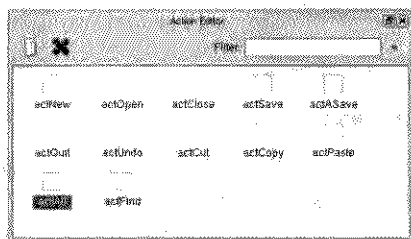


图 4-10 已添加的主菜单的动作

03 将动作编辑器中的动作 `actNew`、`actOpen`、`actSave` 和 `actASave` 拖放到添加的“文件”工具栏上,如图 4-11 所示。

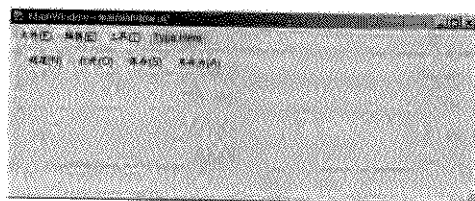


图 4-11 文件工具条按钮

通过同样的方法添加“编辑”工具栏“`editToolBar`”,并将动作 `actUndo`、`actCut`、`actCopy` 和 `actPaste` 拖放到编辑工具栏上,如图 4-12 所示。

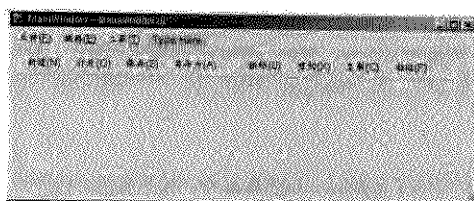


图 4-12 编辑工具条按钮

目前,这些动作是没有图标。为了更直观地在工具栏上显示这些动作,现在给它们添加必要的图标。步骤如下:

01 打开资源编辑器,新建资源文件 `mainwindow.qrc`,保存在 `src/images` 目录下。

02 为资源文件添加图标资源,添加图标资源后的资源编辑器如图 4-13 所示(假设 `src/images` 目录下已经存在这些图标资源)。

03 在动作编辑器中双击“打开”动作 `actOpen`,打开“Edit action”对话框,如图 4-14 所示。

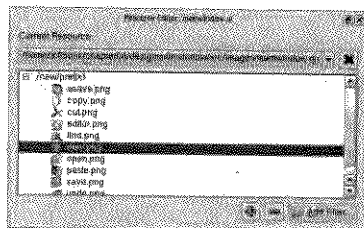


图 4-13 创建主窗口的资源文件



图 4-14 为动作添加图标

04 单击“icon”标签右侧的“...”按钮,弹出“Find icon”对话框,在对话框中选择“Specify resource”单选框,在图标资源列表中选择 `open.png` 图标,单击“确定”按钮,如图 4-15 所示。

05 重新返回到“edit action”对话框,这时候图标已经添加到“icon”标签右侧的按钮上,如图 4-16 所示。

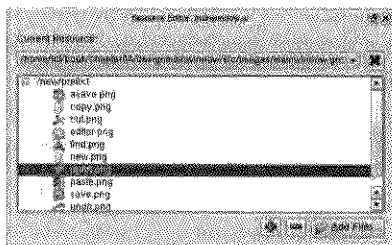


图 4-15 指定图标资源

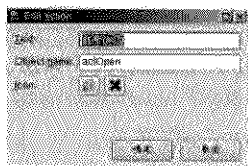


图 4-16 添加图标后的动作

06 在“Edit action”对话框中单击“确定”按钮，图标会自动出现在“文件”|“打开”菜单，“文件”工具栏以及动作编辑器中的 actOpen 动作上。

通过上述步骤，依次为工具栏上的其他动作添加图标，添加图标后的工具栏如图 4-17 所示。添加图标后的动作编辑器如图 4-18 所示。



图 4-17 添加图标后的工具栏

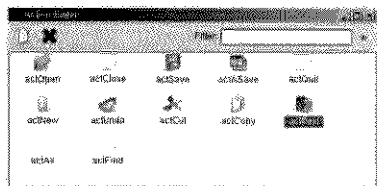


图 4-18 添加图标后的动作编辑器

建好工具栏及其工具按钮后，为了使图标在应用程序运行的时候能够可见，还必须要在 src 目录下的 qmake 工程文件 src.pro 中添加下列语句：

```
RESOURCES += images/mainwindow.qrc
```

修改后的工程文件 src.pro 的内容如下所示。

```
FORMS += mainwindow.ui
HEADERS += mainwindow.h
SOURCES += designmainwindow.cpp \
          mainwindow.cpp
TARGET=../bin/designmainwindow
RESOURCES += images/mainwindow.qrc
```

建立工具栏后，还可以将工具栏拖动到主窗口的左边、右边和下边。将鼠标放在工具栏的左侧且当鼠标变为十字形“+”时按住鼠标左键拖动即可。此外，也可以设置工具栏的一些属性，例如，属性“movable”定义工具栏是否可以被拖动；属性“layoutDirection”定义动作在工具栏上排列的顺序（从左到右或从右到左），等等。

除了利用建立主菜单时添加的动作之外，还可以独立建立新的动作。步骤如下：

01 在动作编辑器中单击“新建”按钮。

02 在出现的“New Action”对话框中输入动作的文本 Text，输入对象名字 Object Name，选择动作的图标（如图 4-14 和图 4-16 所示）。

03 单击“确定”按钮，完成动作的新建。

这些新建的动作可以不在 Qt 设计器中直接使用。保存好 Qt 设计器绘制的主窗口的时候，Qt 设计器便保存了设计的动作、资源文件等。uic 编译器会根据 ui 文件生成创建这些动作对象的代码。之后，可以在应用程序中直接使用代码操控这些动作，比如关联它们的信号和槽、设置动作的一些属性等。

保存绘制的应用程序主窗口，在 KDevelop 中编辑应用程序。在主程序源文件 designmainwindow.cpp 中，在初始化 CMainWindow 对象之前的位置添加如下语句：

```
Q_INIT_RESOURCE(mainwindow);
```

现在编译运行应用程序，可以看到我们添加的工具条及其按钮以及菜单中的图标。

4.2.3 中心部件

在 Qt 设计器中绘制应用程序主窗口的时候，Qt 已经为主窗口自动生成了一个名字为 centralwidget 的、类型为 QWidget 的中心部件，只不过它是空白的，没有任何内容，因此我们很难看到它。不过在 Qt 设计器的对象监视器中，可以一目了然地看到它的存在。主窗口类 QMainWindow 的中心部件可以有多种类型：

- Qt 提供的标准窗口部件，比如 QWidget、QTextEdit 等；
- 用户自定义的窗口部件，比如在第 3 章绘制的查找文件窗口部件；
- 分裂器——QSplitter，QSplitter 作为一个容器可以容纳多个 Qt 窗口部件，此时中心部件是一个包容多个窗口部件的容器；
- 工作空间部件（workspace）——QWorkspace，在一个 MDI 应用程序中，应用程序主窗口的中心部件是一个 QWorkspace 部件；
- 多文档区部件——QMdiArea，它是 Qt 4.3 新增的一个支持多文档的类，用法同 Qworkspace 有些类似。

本章的例子是一个简单文本编辑器，我们把 QTextEdit 作为应用程序主窗口的中心部件：

01 在 Qt 设计器左侧“Widget Box”框中，选择“Input Widgets”中的 QTextEdit 窗口部件，并将它拖放到主窗口中，属性采用默认值。

02 将主窗口放置在一个垂直布局管理器中，添加中心部件后的主窗口如图 4-19 所示。

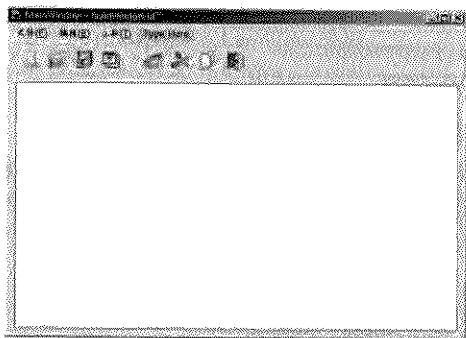


图 4-19 添加中心部件后的主窗口

现在,再次打开对象监视器,可以看到,刚添加的 QTextEdit 窗口部件对象 textEdit 作为中心部件的一个子窗口部件而存在。但这并不影响对 textEdit 窗口部件的使用。

Qt 应用程序的主窗口能够实现对中心部件的自动管理。在主窗口已经存在中心部件的情况下,如果再通过 QMainWindow::setCentralWidget() 函数为主窗口设置一个新的中心部件,那么原来的中心部件将会被主窗口销毁掉。这时候,如果再引用原来的中心部件的话(包括 delete 该对象),将会造成应用程序崩溃。

QTextEdit 是一个功能强大的 Qt GUI 类,是一个高级的所见即所得(What You See is What You Get, WYS / WYG)视图/编辑器(viewer/editor),在处理大文档文件时,做了优化处理,能够快速响应用户的输入。它支持普通文本的显示、编辑,能够加载 HTML 格式的文本,可以显示表格、图片和列表等。此外,QTextEdit 绑定了一些阅读文件时常用的导航键,如表 4-4 所示。

表 4-4 QTextEdit 窗口部件的导航键表

按 键	功 能
Qt::UpArrow	光标移动到上一行
Qt::DownArrow	光标移动到下一行
Qt::LeftArrow	光标向前移动一个字符
Qt::RightArrow	光标向后移动一个字符
PageUp	光标向上移动一页
PageDown	光标向下移动一页
Home	光标移动到当前行的开头
End	光标移动到当前行的结尾
Alt+Wheel	水平滚动页
Ctrl+Wheel	上下快速滚动文本
Ctrl+A	全部选中文本
Ctrl+Home	回到整个文本的第一字符
Ctrl+End	回到整个文本的最后一个字符

此外, QTextEdit 类提供了上下文菜单,菜单内容包括撤消、恢复、剪切、复制、粘贴、删除以及选择全部和插入 Unicode 字符等,如图 4-20 所示。

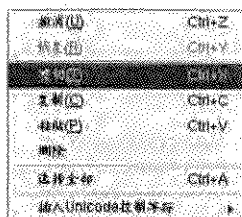


图 4-20 QTextEdit 窗口部件的上下文菜单

功能强大的 QTextEdit 类极大地简化了文本编辑器的实现,无须编程就能够实现文本编辑器常用的快捷键和上下文菜单。

现在重新保存用户界面文件,在 KDevelop 中编译运行应用程序。



4.3 代码创建主窗口

前面详细阐述了如何在 Qt 设计器中绘制应用程序的主窗口，这种方法的特点是直观形象，所见即所得。现在通过手写代码实现应用程序主窗口的创建，有时候手动创建主窗口会比较快捷。之所以提供两种方法创建应用程序的主窗口，主要是为了进行互补。程序员在创建应用程序主窗口的时候，在开发程序的不同阶段，可以有一种选择。如果读者仅仅对 Qt 设计器绘制主窗口感兴趣，完全可以忽略本小节。

首先，创建一个名字为“codemainwindow”的新的 KDevelop 工程（源码路径为 chapter04/codemainwindow）。修改 src 目录下主程序文件 codemainwindow.cpp，内容和 designmainwindow.cpp 的完全相同。

下面依次为工程添加资源文件和主窗口的实现类。

4.3.1 创建资源文件

在新建 KDevelop 工程的 src 目录下创建新的目录 images。接着，将必要的图标资源复制到 images 目录下。然后，在 images 目录下创建资源文件 mainwindow.qrc 文件，编辑该文件添加相应的图标资源。mainwindow.qrc 的内容如下。

```
<RCC>
    <qresource>
        <file>asave.png</file>
        <file>copy.png</file>
        <file>cut.png</file>
        <file>new.png</file>
        <file>open.png</file>
        <file>paste.png</file>
        <file>save.png</file>
        <file>undo.png</file>
    </qresource>
</RCC>
```

4.3.2 定义主窗口类

为 KDevelop 工程添加新的头文件 mainwindow.h，该文件给出了主窗口类 CMainWindow 的定义。文件内容如下所示。

```
// chapter04/codemainwindow/src/mainwindow.h
#ifndef _MAINWINDOW_H_
#define _MAINWINDOW_H_

#include <QMainWindow>
class QTextEdit;

class CMainWindow : public QMainWindow
{
    Q_OBJECT
```

```

public:
    CMainWindow(QWidget* = 0);
private:
    QMenu*      menu_F;
    QMenu*      menu_E;
    QMenu*      menu_T;
    QToolBar*   fileToolBar;
    QToolBar*   editToolBar;
    QAction*    actNew;
    QAction*    actOpen;
    QAction*    actClose;
    QAction*    actSave;
    QAction*    actASave;
    QAction*    actQuit;
    QAction*    actUndo;
    QAction*    actCut;
    QAction*    actCopy;
    QAction*    actFast;
    QAction*    actAll;
    QAction*    actFind;
    QTextEdit*  textEdit;

    void        iniMenu();
    void        iniToolBar();
    void        iniCentralWidget();
};
#endif

```

在类 CMainWindow 的私有区，声明了 3 个指向菜单的指针成员变量，分别指向文件菜单、编辑菜单和工具菜单；定义了 2 个指向工具栏对象的指针，指向文件工具栏和编辑工具栏；定义了 12 个指向动作的指针，指向菜单和工具栏的动作。这些指针将会在关联信号和槽函数等操作中使用。还定义了一个指向 QTextEdit 对象的指针，该 QTextEdit 对象是主窗口的中心部件。

私有函数 iniMenu() 初始化应用程序主窗口的菜单；iniToolBar() 初始化应用程序主窗口的工具栏；iniCentralWidget() 初始化主窗口的中心部件。

下面是主窗口类 CMainWindow 的实现文件 mainwindow.cpp。

```

// chapter04/codemainwindow/src/mainwindow.cpp.
#include <QtGui>
#include "mainwindow.h"

```

程序采用了模块化的设计，因此构造函数比较简单。

```

CMainWindow::CMainWindow(QWidget* parent)
:   QMainWindow(parent)
{
    iniMenu();
    iniToolBar();
    iniCentralWidget();
}

```



```
setWindowTitle(tr("文本编辑器"));
showMaximized();
}
```

接下来，重点看一下类 CMainWindow 的菜单初始化函数和工具栏初始化函数。

```
void CMainWindow::iniMenu()
{
    menu_F = new QMenu(tr("文件(&F)"), this);
    actNew = menu_F->addAction(QIcon(":/new.png"), tr("新建(&N)"));
    actNew->setShortcut(QKeySequence(tr("Ctrl+N")));
```

首先构造一个菜单 QMenu 对象并赋给指针 menu_F，QMenu 构造函数的第一个参数指定了菜单的显示文本；第二个参数指定了菜单对象的父窗口部件。

函数 QMenu::addAction() 添加一个新的动作到菜单的动作列表，并返回该动作对象的指针。此处，该函数具有两个参数：第一个参数构造了一个不具名的、图标为 “:/new.png” 的 QIcon 对象，前面讲过，对于不具名的常量对象的引用是合法的，“:/new.png” 引用 Qt 资源文件中的资源；第二个参数指定了动作对象的显示文本。此外，该函数还有一个只有指定显示文本的重载函数，重载函数只指定了动作的显示文本，没有图标，在下面的代码中有用到。

函数 QAction::setShortcut() 为新加的动作设置快捷键，它的参数是一个不具名的 QKeySequence 对象。此处，QKeySequence 对象构造了一个 “Ctrl+N” 快捷键。

注意，QMenu::addAction(QIcon(":/new.png"), tr("新建(&N)")) 添加的加速键 (accelerator) “N” 与 QAction::setShortcut(QKeySequence(tr("Ctrl+N"))) 添加的快捷键 (shortcut) “Ctrl+N” 的不同之处。为了不至于引起混淆，笔者对加速键和快捷键的使用有意进行了约束：加速键 “N” 必须在文件菜单活动的状态下才有效，即在这种情况下一律使用加速键的概念；而快捷键 “Ctrl+N” 只要在应用程序活动的状态下就可以直接激活（此时，文中使用快捷键描述这种情况）。

接下来，创建 “文件” 菜单的其他动作，以及 “编辑” 菜单和 “工具” 菜单。

```
actOpen = menu_F->addAction(QIcon(":/open.png"), tr("打开(&O)"));
actOpen->setShortcut(QKeySequence(tr("Ctrl+O")));
actClose = menu_F->addAction(tr("关闭(&C)"));
menu_F->addSeparator();
actSave = menu_F->addAction(QIcon(":/save.png"), tr("保存(&S)"));
actSave->setShortcut(QKeySequence(tr("Ctrl+S")));
actASave = menu_F->addAction(QIcon(":/asave.png"), tr("另存为(&A)"));
menu_F->addSeparator();
actQuit = menu_F->addAction(tr("退出(&X)"));

menu_E = new QMenu(tr("编辑(&E)"), this);
actUndo = menu_E->addAction(QIcon(":/undo.png"), tr("撤销(&U)"));
actUndo->setShortcut(QKeySequence(tr("Ctrl+Z")));
menu_E->addSeparator();
actCut = menu_E->addAction(QIcon(":/cut.png"), tr("剪切(&C)"));
actCut->setShortcut(QKeySequence(tr("Ctrl+X")));
actCopy = menu_E->addAction(QIcon(":/copy.png"), tr("复制(&C)"));
actCopy->setShortcut(QKeySequence(tr("Ctrl+C")));
```

```

actPaste = menu_E->addAction(QIcon(":/paste.png"), tr("粘贴(&P)"));
actPaste->setShortcut(QKeySequence(tr("Ctrl+V")));
menu_E->addSeparator();
actAll = menu_E->addAction(tr("全选(&A)"));
actAll->setShortcut(QKeySequence(tr("Ctrl+A")));

menu_T = new QMenu(tr("工具(&T)"), this);
actFind =
    menu_T->addAction(QIcon(":/find.png"), tr("查找文件...(&F)"));

QMenuBar* bar = menuBar();
bar->addMenu(menu_F);
bar->addMenu(menu_E);
bar->addMenu(menu_T);
}

```

函数 `QMainWindow::menuBar()` 获取主窗口的菜单栏。如果主窗口的菜单栏已经存在，该函数将返回菜单栏的指针；否则该函数将创建并返回一个新的菜单栏。

函数 `QMainWindow::addMenu()` 将前面定义好的菜单添加到主窗口的菜单栏上。

```

void QMainWindow::initToolBar()
{
    fileToolBar = new QToolBar(this);
    fileToolBar->setAllowedAreas(Qt::AllToolBarAreas);
    fileToolBar->setOrientation(Qt::Horizontal);
    fileToolBar->addAction(actNew);
    fileToolBar->addAction(actOpen);
    fileToolBar->addAction(actSave);
    fileToolBar->addAction(actASave);
}

```

在初始化工具栏函数 `initToolBar()` 中，首先调用 `QToolBar` 的构造函数创建一个“文件”工具栏。函数 `QToolBar::setAllowedAreas()` 设置工具栏可放置的位置，共有 6 个选项（`Qt::ToolBarAreas` 枚举类型）：

- `Qt::LeftToolBarArea`，主窗口的左侧；
- `Qt::RightToolBarArea`，主窗口的右侧；
- `Qt::TopToolBarArea`，主窗口的顶部；
- `Qt::BottomToolBarArea`，主窗口的底部；
- `Qt::AllToolBarAreas`，可以放在主窗口四个方向的任何一个；
- `Qt::NoToolBarArea`，没有可放置工具条的区域。

工具栏放置区域的前四个选择可以通过位或操作运算符“|”进行任意的组合。

函数 `QToolBar::setOrientation()` 设置工具栏图标的排列方向。事实上，Qt 的工具栏的方向是由主窗口 `QMainWindow` 对象自动管理的，当一个工具栏被放置在主窗口的顶部时，就决定了工具栏的方向是水平的。该函数的参数由两个选项：

- `Qt::Horizontal`，水平方向；
- `Qt::Vertical`，垂直方向。

此处，无论设置哪个选项，都不会影响工具栏在当前放置区域的方向。

`QToolBar::addAction()` 将动作添加到相应的工具栏上，该函数继承自 `QToolBar` 类的父类 `QWidget`。



下面完成“编辑”工具栏的创建。

```
editToolBar = new QToolBar(this);
editToolBar->setAllowedAreas(Qt::AllToolBarAreas);
editToolBar->setOrientation(Qt::Horizontal);
editToolBar->addAction(actUndo);
editToolBar->addAction(actCut);
editToolBar->addAction(actCopy);
editToolBar->addAction(actPaste);

addToolBar(Qt::TopToolBarArea, fileToolBar);
addToolBar(Qt::TopToolBarArea, editToolBar);
}
```

最后，QMainWindow::addToolBar()将“文件”工具栏和“编辑”工具栏添加到主窗口。它的一个参数指定了工具栏放置的区域；第二个参数指定了被放置的工具栏对象。

```
void CMainWindow::iniCentralWidget()
{
    textEdit = new QTextEdit(this);
    setCentralWidget(textEdit);
}
```

iniCentralWidget()函数构造一个 QTextEdit 对象，并将它设置为应用程序主窗口的中心部件。

修改 KDevelop 工程的 qmake 工程文件 src.pro，加入资源选项：

```
RESOURCES += ./images/mainwindow.qrc
```

现在编译运行应用程序，它的显示效果和 Qt 设计器绘制的界面效果基本上是相同的，如图 4-21 所示。

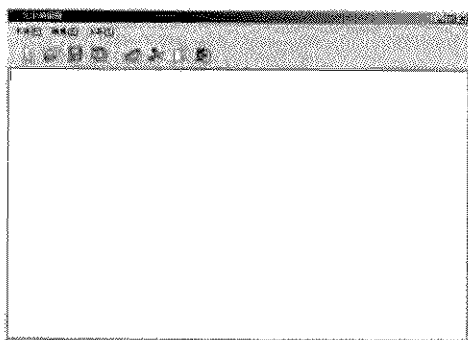


图 4-21 主窗口的运行效果图

4.4 锚接部件

锚接部件 QDockWidget 继承自 Qt 的基础窗口部件 QWidget，它可以作为 QMainWindow 的一个窗口部件停靠在应用程序主窗口的内部（上下左右四个位置都可以，见应用程序主窗口的布局框架），也可以作为一个独立的窗口浮动在应用程序主窗口的上面。它经常被作为工具面板或工具窗口来使用。

比如, Qt 设计器中的对象监视器、属性编辑器以及动作编辑器等都是以前接部件的形式存在的。

锚接部件 QDockWidget 为设计应用程序主窗口界面提供了更加灵活方便、易用直观的操作模式。

目前, Qt 设计器对锚接部件的支持不是很理想。比如, 当通过 Qt 设计器为主窗口添加一个锚接部件后, 运行应用程序, 锚接部件能够浮动起来, 但不能像窗口一样进行拖动。通过使用代码添加的锚接部件是没有这些问题的。同样, 在 Qt 设计器中也无法编辑主窗口的状态栏, 为状态栏添加相应的窗口部件。

在简单文本编辑器例子中, 通过手写代码添加主窗口的锚接部件 (及其子窗口部件) 和状态栏的显示窗口部件。下面把第 3 章设计的查找文件窗口部件加入到锚接部件容器中, 在简单文本编辑器中实现对文件的查找。

以 Qt 设计器绘制的主窗口为例, 为主窗口添加锚接部件。在代码创建的主窗口界面中实现锚接部件的添加是相同的。

首先, 将第 3 章中设计的 CFindFileForm 类的头文件 findfileform.h、实现文件 findfileform.cpp 和 ui 文件 findfileform.ui 加入到当前的 KDevelop 工程中来。有两种方法:

- 将这些文件复制到当前工程的 src 目录下, 然后修改 qmake 工程文件;
- 在 KDevelop 的“QMake 管理器”选项卡中, 在相应的“HEADERS”、“SOURCES”和“FORMS”选项上单击鼠标右键, 在出现的对话框中选择需要添加的文件。对话框中最下面的下拉框选择“复制文件”选项, 如图 4-22 所示。

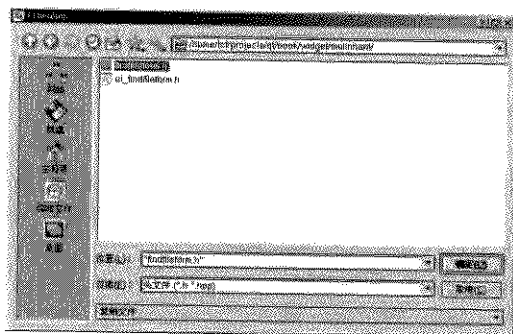


图 4-22 将以存在的文件加入当前工程

修改主窗口类 CMainWindow 的头文件, 添加指向锚接部件对象的指针和初始化函数。代码如下所示。

```
// chapter04/designmainwindow/src/mainwindow.h
#ifndef _MAINWINDOW_H_
#define _MAINWINDOW_H_

#include "ui_mainwindow.h"
class QLabel;
class CMainWindow : public QMainWindow,
                   public Ui::MainWindow
{
    Q_OBJECT
```



```

public:
    CMainWindow(QWidget* = 0);

private:
    QDockWidget* dockWidget;

    void      iniDockWidget();
};
#endif

```

修改主窗口类的实现文件 `mainwindow.cpp`，将类 `CFindFileForm` 的头文件包含进来。修改主窗口类的构造函数，添加对锚接部件初始化函数 `iniDockWidget()` 的调用。

锚接部件初始化函数如下所示。

```

// chapter04/designmainwindow/src/mainwindow.cpp.
void CMainWindow::iniDockWidget()
{
    CFindFileForm* findFileForm = new CFindFileForm;
    dockWidget = new QDockWidget(tr("查找文件"), this);
    dockWidget->setAllowedAreas(Qt::RightDockWidgetArea);
    dockWidget->setFeatures(QDockWidget::AllDockWidgetFeatures);
    dockWidget->setFloating(false);
    dockWidget->setWidget(findFileForm);
    dockWidget->setVisible(true);

    addDockWidget(Qt::RightDockWidgetArea, dockWidget);
}

```

在函数 `iniDockWidget()` 函数中，首先构造 `CFindFileForm` 对象和锚接部件对象。`QDockWidget::setAllowedAreas()` 函数设置锚接部件在主窗口中可以停靠的位置，它的参数是一个 `Qt::DockWidgetAreas` 枚举类型的值（具体含义与 `Qt::ToolBarAreas` 的枚举值类似）：

- `Qt::LeftDockWidgetArea`，主窗口的左侧；
- `Qt::RightDockWidgetArea`，主窗口的右侧；
- `Qt::TopDockWidgetArea`，主窗口的顶部；
- `Qt::BottomDockWidgetArea`，主窗口的底部；
- `Qt::AllDockWidgetAreas`，主窗口的任何一个方位；
- `Qt::NoDockWidgetArea`，不在任何一个位置。

函数 `QDockWidget::setFloating()` 设置锚接部件初始显示的时候是否要浮动在主窗口的上面。在此，传递了一个 `false` 参数，即锚接部件第一次显示的时候将停靠在主窗口中。

然后，将查找文件窗口部件放置到锚接部件中，并设置锚接部件为可见的。

最后，函数 `QMainWindow::addDockWidget()` 将锚接部件加入到应用程序主窗口中。

OK，现在编译运行应用程序。显示效果如图 4-23 所示。

在主窗口中，当单击查找文件窗口部件的“关闭”按钮时，查找文件窗口部件会被关闭掉，而锚接部件仍然可见，只是一个空的容器而已。这不是我们想要的运行效果。作为一个锚接部件例子，目前已经详细说明了 `QDockWidget` 的使用，如果读者感兴趣的话，可以进一步完善这个功能。

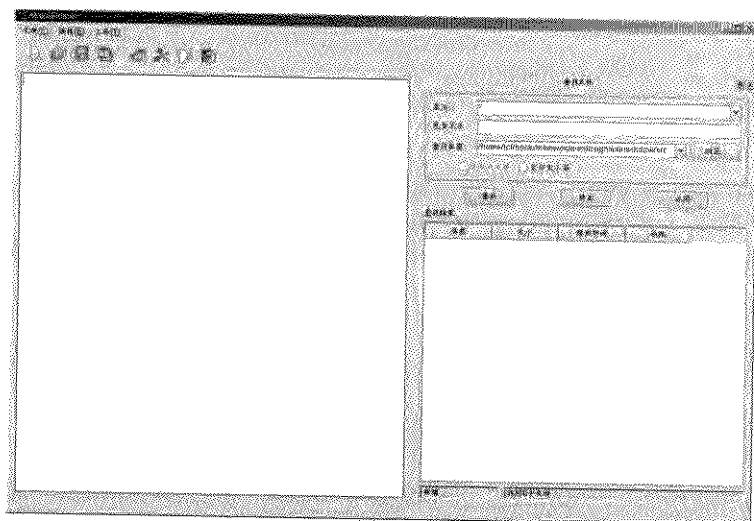


图 4-23 加入锚接部件后的主窗口界面

4.5 状态栏

GUI 图形用户界面的状态栏一般用来显示当前系统的状态信息以及一些提示信息。Qt 主窗口的状态栏可以添加任意的 Qt 窗口部件(或 Qt 提供的标准窗口部件或自定义的继承自 QWidget 的窗口部件)。

同样, 以 Qt 设计器绘制的主窗口为例, 为主窗口添加状态栏及其显示窗口部件。

现在为应用程序主窗口创建一个状态栏, 主要显示文本编辑器的一些简单的编辑状态。重新修改主窗口 CMainWindow 类的头文件 mainwindow.h, 如下所示。

```
// chapter04/designmainwindow/src/mainwindow.h
#ifndef _MAINWINDOW_H_
#define _MAINWINDOW_H_

#include "ui_mainwindow.h"
class QLabel;

class CMainWindow : public QMainWindow,
                   public Ui::MainWindow
{
    Q_OBJECT
public:
    CMainWindow(QWidget* = 0);

private:
    QDockWidget*   dockWidget;
    QLabel*        label1;
```

```

    QLabel*      label2;

    void          iniDockWidget();
    void          iniStatusBar();
};
#endif

```

在头 `mainwindow.h` 文件中，添加一行 `QLabel` 类的传递声明：声明了两个指向标签 `QLabel` 对象的指针 `label1` 和 `label2`；声明了状态栏的初始化函数 `iniStatusBar()`。

在 `CMainWindow` 类的实现文件 `mainwindow.cpp` 中，添加初始化状态栏函数 `iniStatusBar()` 的实现。

```

void CMainWindow::iniStatusBar()
{
    QStatusBar* bar = statusBar();
    label1 = new QLabel;
    label1->setMinimumSize(200, 25);
    label1->setFrameShape(QFrame::WinPanel);
    label1->setFrameShadow(QFrame::Sunken);
}

```

函数 `QMainWindow::statusBar()` 获取应用程序主窗口的状态栏。如果状态栏已经存在，则返回指向主窗口的状态栏的指针；如果状态栏还没有建立，该函数将新建一个状态栏并添加到主窗口上，然后返回新建状态栏的指针。

函数 `QLabel::setMinimumSize()` 设置标签的最小尺寸，以便能够有足够的空间显示信息。在这种情况下，标签的大小是不会伸缩的，除非标签确实无法存放需要显示的全部信息。

`QLabel::setFrameShape()` 函数设置标签的形状，该函数继承自 `QFrame` 类。`QFrame` 类定义了 7 种形状：

- `QFrame::NoFrame`，没有边框，此时 `QFrame` 什么也不绘制；
- `QFrame::Box`，在显示内容周围绘制一个方框；
- `QFrame::Panel`，绘制一个面板（panel），使得显示内容凸起或凹陷；
- `QFrame::StyledPanel`，绘制一个方形的面板，但要依赖于目前使用的 GUI 类型；显示内容凹陷或凸起；
- `QFrame::HLine`，绘制一条水平线；
- `QFrame::VLine`，绘制一条垂直线；
- `QFrame::WinPanel`，绘制一个凸起或凹陷的类似于 Windows 95 中的方形面板，线宽为两个像素。提供该类型的目的主要是为了实现兼容性。

在此，`QLabel::setFrameShape()` 的参数采用 `QFrame::WinPanel`，它的 3D 效果比较明显。

函数 `QLabel::setFrameShadow()` 设置标签外框的阴影，该函数继承自 `QFrame` 类。`QFrame` 类定义了 3 种阴影模式：

- `QFrame::Plain`，窗口部件的内容和外框同该窗口部件的父窗口部件处在同一水平（即没有凹凸感、立体感），在没有 3D 效果的情况下，设置该阴影模式的窗口部件使用画板的前景色绘制；
- `QFrame::Raised`，窗口部件的边框和内容是凸起的，具有 3D 效果；
- `QFrame::Sunken`，窗口部件的边框和内容是凹陷的，具有 3D 效果。

此处，设置标签 `label1` 的显示效果为 `QFrame::Sunken`。注意，函数 `QLabel::setFrameShadow()` 的效果要受到 `QFrame::setFrameShape()` 函数参数的影响。

接下来，使用同样的方法添加第二个标签 QLabel 对象 label2。

```
label2 = new QLabel;
label2->setMinimumSize(200, 25);
label2->setFrameShadow(QFrame::Sunken);
label2->setFrameShape(QFrame::WinPanel);

bar->addWidget(label1);
bar->addWidget(label2);
}
```

最后，QStatusBar::addWidget()函数将两个标签对象 label1 和 label2 添加到主窗口的状态栏中。该函数具有两个形参：第一个参数指定了需要添加的窗口部件；第二个参数指定了一个伸缩因子，当状态栏伸长或压缩的时候，布局管理器用该因子来计算窗口部件的大小，以便使被添加的窗口部件的尺寸适合于新的情况。第二个参数采用默认值 0，它表示在窗口部件能够完全显示其内容的前提下，该窗口部件的尺寸是最小的。

在构造函数中添加对函数 iniStatusBar()的调用。

重新编译、运行应用程序，显示效果如图 4-24 所示。

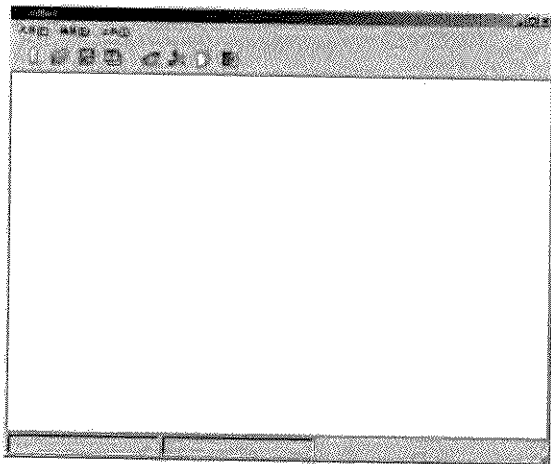


图 4-24 加入状态条后的主窗口界面

4.6 实现文本编辑器功能

现在基本上已经创建好了应用程序的主窗口界面，包括主窗口的菜单、工具栏、中心部件、锚接部件以及状态栏。接下来，实现文本编辑器的基本功能，包括新建文件、打开文件、保存文件，实现编辑器的操作撤销，以及文本的剪切、复制、粘贴和文本全选功能等。

还是以 Qt 设计器绘制的主窗口为例，实现上述编辑器功能。在代码创建的主窗口界面中实现文本编辑器的功能是相同的。因为在定义成员变量的时候采用和 Qt 设计器中相同的名字，因此，只要将功能实现代码复制到相应的头文件和实现文件中就可以了。

首先，分析一下主窗口类 CMainWindow 的定义文件 mainwindow.h，内容如下。

```
// chapter04/design/mainwindow/src/mainwindow.h
#ifndef _MAINWINDOW_H_
#define _MAINWINDOW_H_

#include "ui_mainwindow.h"
class QLabel;

class CMainWindow : public QMainWindow,
                   public Ui::MainWindow
{
    Q_OBJECT
public:
    CMainWindow(QWidget* = 0);

private:
    QDockWidget*   dockWidget;
    QLabel*        label1;
    QLabel*        label2;
    bool           isUntitled;
    QString        curFile;
    enum{MaxRecentFiles = 9};
    QAction*       recentFileActs[MaxRecentFiles];
    QAction*       separatorAct;

    void           iniDockWidget();
    void           iniStatusBar();
    void           iniConnect();
    void           setCurrentFile(const QString&);
    bool           saveFile(const QString&);
    bool           loadFile(const QString&);
    void           maybeSave();
    void           updateRecentFiles();
}
```

在类 CMainWindow 的私有区，新声明了 5 个成员变量，布尔型变量 isUntitled 记录当前正在编辑的文本是否已经保存在硬盘上；字符串变量 curFile 保存当前编辑的文本文件的文件名；枚举变量 MaxRecentFiles 指定了最近文件列表的最多文件数目；数组 recentFileActs[] 存放预先初始化好的 QAction 对象，以备在文件菜单上显示最近的文件列表；动作对象指针 separatorAct 指向一个菜单间隔线(separator)，当文本编辑器显示最近打开的文件时，在文件菜单上显示间隔线，否则隐藏。

私有函数 iniConnect()完成所有窗口部件的信号和槽的关联操作；函数 setCurrentFile()设置文本编辑器的编辑状态；函数 saveFile()保存指定文件名的文件；函数 loadFile()从指定的位置读取指定的文件到文本编辑器；函数 maybeSave()判断是否要保存文件，如果是则进行保存；函数 updateRecentFiles()更新文本编辑器中文件菜单上的最近文件列表。

```
private slots:
    void doCursorChanged();
```

```

void doNew();
void doOpen();
void doClose();
void doSave();
void doASave();
void doQuit();
void doUndo();
void doCut();
void doCopy();
void doPast();
void doAll();
void doFind();
void doModified();
void openRecentFile();
};
#endif

```

头文件定义了 14 个私有槽函数，实现相应的菜单和工具栏功能。槽函数 doCursorChanged() 实现光标移动时动态显示光标在编辑器状态中的位置；槽函数 doModified() 完成编辑器状态变化时的实时状态显示；槽函数 openRecentFile() 打开最近文件列表指定的文件。

接下来，看一下类 CMainWindow 的实现文件 mainwindow.cpp 的内容。

```

// chapter04/designmainwindow/src/mainwindow.cpp.
#include <QtGui>

#include "mainwindow.h"
#include "findfileform.h"

CMainWindow::CMainWindow(QWidget* parent)
: QMainWindow(parent)
{
    setupUi(this);

    iniDockWidget();
    iniStatusBar();
    iniConnect();
    updateRecentFiles();
    showMaximized();

    isUntitled = true;
    curFile = tr("untitled");
    setWindowTitle(curFile + " [*]");
}

```

类 CMainWindow 的构造函数实现文本编辑器应用程序主窗口的初始化，包括初始化锚接部件、主窗口状态栏和关联所有必需的信号和槽。

在文本编辑器启动的时候，调用函数 updateRecentFiles() 完成文件菜单上的最近文件列表的更新。初始化成员变量 isUntitled 为 true，表明新文档从未保存过。



函数 `setWindowTitle()` 设置窗口的标题。标题文本中的 “[*]” 指示当文档被修改时，窗口标题中的修改标识 “*” 出现的位置。必须指定 “[*]”，否则当文档改变时，窗口的标题不会出现修改标识 “*”。

```
void CMainWindow::iniDockWidget()
{
    CFindFileForm* findFileForm = new CFindFileForm;
    dockWidget = new QDockWidget(tr("查找文件"), this);
    dockWidget->setAllowedAreas(Qt::RightDockWidgetArea);
    dockWidget->setFeatures(QDockWidget::AllDockWidgetFeatures);
    dockWidget->setFloating(false);
    dockWidget->setWidget(findFileForm);
    dockWidget->setVisible(true);

    addDockWidget(Qt::RightDockWidgetArea, dockWidget);
}
```

函数 `iniDockWidget()` 实现对锚接部件的初始化，完成查找文件功能的初始化。前面已经详细分析了该函数的实现细节。

```
void CMainWindow::iniStatusBar()
{
    QStatusBar* bar = statusBar();
    label1 = new QLabel;
    label1->setMinimumSize(200, 25);
    label1->setFrameShadow(QFrame::Sunken);
    label1->setFrameShape(QFrame::WinPanel);
    label2 = new QLabel;
    label2->setMinimumSize(200, 25);
    label2->setFrameShadow(QFrame::Sunken);
    label2->setFrameShape(QFrame::WinPanel);
    bar->addWidget(label1);
    bar->addWidget(label2);
}
```

函数 `iniStatusBar()` 完成主窗口状态栏的初始化。

```
void CMainWindow::iniConnect()
{
    connect(textEdit, SIGNAL(cursorPositionChanged()),
           this, SLOT(doCursorChanged()));
    connect(actNew, SIGNAL(triggered()),
           this, SLOT(doNew()));
    connect(actOpen, SIGNAL(triggered()),
           this, SLOT(doOpen()));
    connect(actClose, SIGNAL(triggered()),
           this, SLOT(doClose()));
    connect(actSave, SIGNAL(triggered()),
           this, SLOT(doSave()));
    connect(actASave, SIGNAL(triggered()),
           this, SLOT(doASave()));
}
```

```

connect(actQuit, SIGNAL(triggered()),
        this, SLOT(doQuit()));
connect(actUndo, SIGNAL(triggered()),
        this, SLOT(doUndo()));
connect(actCut, SIGNAL(triggered()),
        this, SLOT(doCut()));
connect(actCopy, SIGNAL(triggered()),
        this, SLOT(doCopy()));
connect(actPaste, SIGNAL(triggered()),
        this, SLOT(doPast()));
connect(actAll, SIGNAL(triggered()),
        this, SLOT(doAll()));
connect(actFind, SIGNAL(triggered()),
        this, SLOT(doFind()));
connect(textEdit->document(), SIGNAL(contentsChanged()),
        this, SLOT(doModified()));
separatorAct = menu_F->insertSeparator(actQuit);
separatorAct->setVisible(false);
for (int i = 0; i < MaxRecentFiles; ++i)
{
    recentFileActs[i] = new QAction(this);
    menu_F->insertAction(separatorAct, recentFileActs[i]);
    recentFileActs[i]->setVisible(false);
    connect(recentFileActs[i], SIGNAL(triggered()),
            this, SLOT(openRecentFile()));
}
}

```

函数 `iniConnect()` 完成相关信号和槽的关联。对于编辑器的有些功能，可以通过直接将 Qt 窗口部件提供的信号和槽关联起来就可以了。比如将 `QAction` 对象 `actCopy` 的 `triggered()` 信号同 `QTextEdit` 对象 `textEdit` 的槽 `copy()` 直接关联起来完成选中文本的复制操作。但为了便于解释代码，将 `actCopy` 的 `triggered()` 信号同 `CMainWindow` 对象的槽 `doCopy()` 关联起来，然后再在槽 `doCopy()` 中调用槽函数 `QText::copy()`。

在 `iniConnect()` 函数的最后，初始化文件菜单的最近文件列表的动作，将它们插入到文件菜单中，并隐藏这些动作。

```

void CMainWindow::doCursorChanged()
{
    int pageNum = textEdit->document()->pageCount();
    const QTextCursor cursor = textEdit->textCursor();
    int colNum = cursor.columnNumber();
    int rowNum = textEdit->document()->blockCount();
    label1->setText(tr("%1 页   %3 列").arg(pageNum).arg(colNum));
}

```

槽函数 `doCursorChanged()` 接收文本编辑器中光标的位置变化时 `QTextEdit` 对象发出的信号 `QTextEdit::cursorPositionChanged()`，它完成主窗口状态栏中相应显示信息的更新。

函数 `QTextEdit::document()` 获取 `QTextEdit` 对象的底层 `QTextDocument` 对象的指针并返回。



QTextDocument 是一个结构化的富文本 (rich text) 文档的容器, 提供了对样式文本 (styled text) 和多种文档元素的支持, 比如列表 (lists)、表格 (tables)、框 (frames) 和图片; 提供了一些方便的操作, 比如撤销 undo、恢复 redo 等。QTextDocument 既可以为 QTextEdit 所使用, 也可以单独使用。在构造一个 QTextEdit 对象的时候, 也将会构造一个 QTextDocument 对象, 通过函数 QTextEdit::document() 获取该 QTextDocument 对象的指针引用。QTextDocument::pageCount() 函数获取文本文档的页数。

函数 QTextEdit::textCursor() 获取当前可见光标 QTextCursor 对象的一个拷贝。QTextCursor 类封装了访问和编辑 QTextDocument 的 API 接口。

QTextCursor::columnNumber() 函数获取光标所在的列位置。

QTextDocument::blockCount() 函数获取文档的块数。因为本例子只是一个简单的文本编辑器, 没有图片、表格等其他文档元素, 因此该函数返回的也是文档的实际行数。

最后, 将获得文档的信息显示在状态栏上的 label1 中。

```
void CMainWindow::doFind()
{
    dockWidget->show();
    dockWidget->setFloating(false);
}
```

槽函数 doFind() 显示锚接部件, 并将它停靠在应用程序主窗口内。

```
void CMainWindow::doNew()
{
    maybeSave();
    isUntitled = true;
    curFile = tr("Untitled");
    setWindowTitle(curFile + " [*]");
    textEdit->clear();
    textEdit->setVisible(true);
    setWindowModified(false);
}
```

槽函数 doNew() 完成创建新文本文件的功能。首先, 它调用 maybeSave() 函数对先前编辑过的文本的进行保存。然后, 设置 isUntitled 成员变量的值为 true, 表示新文档从来没有保存过。最后, 设置应用程序窗口的标题和修改状态。

函数 QTextEdit::clear() 清空文本编辑框的内容, 同时也清除了撤销和恢复的历史, 并设置文档的修改状态为未编辑的 (此时 QTextEdit::isModified() 返回 false)。

函数 QMainWindow::setWindowModified() 设置应用程序窗口的修改状态, 如果参数为 true, 则窗口标题将会出现修改标识 “*”。

```
void CMainWindow::maybeSave()
{
    if(textEdit->document()->isModified())
    {
        QMessageBox box;
        box.setWindowTitle(tr("警告"));
        box.setIcon(QMessageBox::Warning);
        box.setText(tr("文档没有保存, 是否保存? "));
    }
}
```



```

        box.setStandardButtons(QMessageBox::Yes
                               | QMessageBox::No);
        if (box.exec() == QMessageBox::Yes)
        {
            doSave();
        }
    }
}

```

函数 `maybeSave()` 实现文件的保存。它首先判断文档在最后一次保存之后是否有修改，如果没有修改该函数返回；如果修改了，它会弹出一个对话框，提示用户是否要保存。如果用户需要保存文件，则调用槽函数 `doSave()` 完成文本文件的真正保存。

```

void CMainWindow::doOpen()
{
    QString fileName = QFileDialog::getOpenFileName(this);
    if (!fileName.isEmpty())
    {
        maybeSave();
        if (loadFile(fileName))
        {
            label2->setText(tr("已经读取"));
        }
    }
    textEdit->setVisible(true);
}

```

槽函数 `doOpen()` 响应用户的打开操作，打开一个新的文件。它首先弹出一个打开文件对话框，如果用户选择了要打开的文件，该函数将会保存当前正在编辑的文件（如果还没有及时保存的话），然后调用 `loadFile()` 读取指定的文件，并设置状态栏的显示信息为“已经读取”。

```

bool CMainWindow::loadFile(const QString& fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly | QFile::Text))
    {
        QMessageBox::warning(this, tr("读取文件"),
                             tr("无法读取文件 %1:\n%2.")
                             .arg(fileName)
                             .arg(file.errorString()));
        return false;
    }
    QTextStream in(&file);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    textEdit->setText(in.readAll());
    QApplication::restoreOverrideCursor();
    setCurrentFile(fileName);

    return true;
}

```



loadFile()函数读取指定的文件内容并显示在应用程序主窗口的文本编辑框内。

函数 `QApplication::setOverrideCursor()` 设置应用程序的鼠标状态。当鼠标移动到应用程序的任何一个窗口部件上时,鼠标的形状都是新设置的鼠标形状。一般当应用程序向用户显示一个特定的状态时,需要调用该函数设置鼠标的形状,比如比较耗时的操作。它具有一个 `QCursor` 类型的形参。在此,传递实参 `Qt::WaitCursor` 给 `QCursor` 的构造函数,它将构造一个“等待”形状的鼠标对象。

函数 `QApplication::restoreOverrideCursor()` 将鼠标形状恢复到调用函数 `QApplication::setOverrideCursor()` 之前的状态。

最后,调用 `setCurrentFile()` 设置当前文本编辑器的状态。函数 `setCurrentFile()` 的实现如下所示。

```
void CMainWindow::setCurrentFile(const QString& fileName)
{
    curFile = QFileInfo(fileName).canonicalFilePath();
    isUntitled = false;
    setWindowTitle(curFile + "[*]");
    textEdit->document()->setModified(false);
    setWindowModified(false);

    QSettings settings("709", "SDI example");
    QStringList files = settings.value("recentFiles").toStringList();
    files.removeAll(fileName);
    files.prepend(fileName);
    while (files.size() > MaxRecentFiles)
        files.removeLast();
    settings.setValue("recentFiles", files);

    updateRecentFiles();
}
```

函数 `QFileInfo::canonicalFilePath()` 返回一个标准的路径(不包含符号链接、当前目录“.”和父目录“..”的路径)。在一个没有符号链接的系统上该函数将会和函数 `QFileInfo::absoluteFilePath()` 返回相同的字符串。

`QSettings` 类提供了平台无关的、永久保存应用程序配置项的功能。它使用<键, 值>对的模式保存应用程序的数据。`QSettings` 的键有点类似于文件系统的路径,可以通过类似于路径的方式指定子键(比如“SDI example/currentFileList”),也可以使用 `beginGroup()` 和 `endGroup()` 函数开始或结束一个键(子键)。例如,

```
QStringList fileList;
fileList << "file1" << "file2";
settings.setValue("SDI example/currentFileList", fileList);
或者
QStringList fileList;
fileList << "file1" << "file2";
settings.beginGroup("SDI example");
settings.setValue("currentFileList", fileList);
settings.endGroup();
```

在默认情况下, `QSettings` 将会保存应用程序的数据到一个平台相关的位置。在 Windows 系统,它使用系统注册表;在 Unix 系统,它将数据保存在文本文件(用户目录下的 `config` 文件。例如,作者使用的

是红旗 Linux 5.0 工作站系统, 运行该应用程序后, Qt 会将这些数据保存在/home/pcf/config 文件中); 在 Mac 系统, 它使用 Core Foundation Preferences API。

QSettings 构造函数的参数指定了组织机构的名字和应用程序的名字。QSetting 将会使用这些字符串信息并根据特定的平台查找应用程序保存的数据。

QSettings::value()函数取出应用程序原先保存的数据, 并保存在一个 QStringList 列表中。完成最近文件列表的更新后, 最后 QSettings::setValue()将数据重新保存, 原来的数据将会被覆盖。

最后, updateRecentFiles()更新文件菜单中的最近文件列表, 代码实现如下。

```
void CMainWindow::updateRecentFiles()
{
    QSettings settings("709", "SDI example");
    QStringList files = settings.value("recentFiles").toStringList();

    int numRecentFiles = qMin(files.size(), (int)MaxRecentFiles);
    for (int i = 0; i < numRecentFiles; ++i)
    {
        QString text = tr("&%1 %2").arg(i + 1).arg(files[i]);
        recentFileActs[i]->setText(text);
        recentFileActs[i]->setData(files[i]);
        recentFileActs[i]->setVisible(true);
    }
    for (int i = numRecentFiles; i < MaxRecentFiles; ++i)
        recentFileActs[i]->setVisible(false);

    separatorAct->setVisible(numRecentFiles > 0);
}
```

updateRecentFiles()函数中, 首先获取应用程序保存的最近文件列表, 然后根据列表中的文件数目, 依次设置文件菜单中的动作。

qMin()是 Qt 提供的一个全局的模板函数, 它返回两个数值中的最小值。

函数 QAction::setData()设置动作的内部数据, 通过 QAction::data()可以重新获得内部数据的内容。设置的数据内容是一个文件的绝对路径, 当打开最近列表中的文件时将用到该数据。

最后, 如果最近文件列表非空, 那么显示菜单间隔线 separatorAct 对象; 否则, 置为不可见。

```
void CMainWindow::doClose()
{
    maybeSave();
    textEdit->setVisible(false);
}
```

槽函数 doClose()接收用户的“关闭”操作发出的信号, 关闭当前显示的文本文档。

```
void CMainWindow::doSave()
{
    if (isUntitled)
    {
        doASave();
    }
}
```



```
else
{
    saveFile(curFile);
}
```

槽函数 `doSave()` 接收用户的“保存”操作发出的信号，完成文件的保存。

```
void CMainWindow::doASave()
{
    QString fileName =
        QFileDialog::getSaveFileName(this, tr("另存为"), curFile);
    if (!fileName.isEmpty())
    {
        saveFile(fileName);
    }
}
```

槽函数 `doASave()` 接收用户的“另存为”操作发出的信号，将文本文档保存到一个新的文件中。

```
void CMainWindow::doQuit()
{
    doClose();
    qApp->quit();
}
```

槽函数 `doQuit()` 接收用户的“退出”文本编辑器操作发出的信号，保存文件并退出系统。

在一个应用程序中，只能创建一个 `QApplication` 对象。变量 `qApp` 是一个全局的指向 Qt 应用程序的指针，它等价于调用函数 `QCoreApplication::instance()`。

```
void CMainWindow::doUndo()
{
    textEdit->undo();
}
```

槽函数接收用户撤销操作发出的信号，完成文本编辑器的撤销操作。该函数直接调用 `QTextEdit` 类提供的函数 `undo()`。

```
void CMainWindow::doCut()
{
    textEdit->cut();
}
```

槽函数 `doCut()` 完成文本的剪切功能。槽函数 `QTextEdit::cut()` 将会将选中的文本内容剪切到剪贴板。

```
void CMainWindow::doCopy()
{
    textEdit->copy();
}
```

槽函数 `doCopy()` 完成文本的复制功能。槽函数 `QTextEdit::copy()` 将会将选中的文本内容复制到剪贴板。

```
void CMainWindow::doPaste()
{
    textEdit->paste();
}
```

槽函数 `doPaste()` 完成文本的粘贴功能。槽函数 `QTextEdit::paste()` 将剪贴板中的文本粘贴到光标所在的位置。

```
void CMainWindow::doAll()
{
    textEdit->selectAll();
}
```

槽函数 `doAll()` 选中文本编辑器中的全部文本内容。

```
bool CMainWindow::saveFile(const QString& fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::WriteOnly | QFile::Text))
    {
        QMessageBox::warning(this,
                               tr("保存文件"),
                               tr("无法保存文件 %1:\n%2.")
                               .arg(fileName)
                               .arg(file.errorString()));
        return false;
    }

    QTextStream out(&file);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    out << textEdit->toPlainText();
    QApplication::restoreOverrideCursor();
    setCurrentFile(fileName);
    label2->setText(tr("已经保存"));

    return true;
}
```

函数 `saveFile()` 实现对指定文件的保存。

函数 `QTextEdit::toPlainText()` 将文本编辑框中的文本内容转换为普通文本格式并返回转换后的文本内容。

```
void CMainWindow::doModified()
{
    setWindowModified(textEdit->document()->isModified());
    label2->setText(tr("正在修改"));
}
```

槽函数 `doModified()` 接收 `QTextEdit` 对象 `textEdit` 的信号 `textChanged()`，完成文本编辑器状态的修改和显示。当文本编辑框 `QTextEdit` 的内容改变时，`QTextEdit` 对象会发出信号 `textChanged()`。



```
void CMainWindow::openRecentFile()  
{  
    QAction *action = qobject_cast<QAction *>(sender());  
    if (action)  
        loadFile(action->data().toString());  
}
```

槽函数 `openRecentFile()` 接收用户选择文件菜单中的打开最近文件操作时发出的信号，打开最近文件列表指定的文件。

现在，重新编译、运行应用程序。

4.7 多 文 档

到目前为止，已经实现了一个简单的文本编辑器。用户在某一时刻，只能阅读、编辑一个文档，极为不方便。下面，在前面的单文档文本编辑器的基础上，实现文本编辑器的多文档功能。

实现文本编辑器的多文档，可以有多种方法：

- 在一个应用程序中实例化多个主窗口，即当打开或新建文档的时候，文本编辑器应用程序新建一个主窗口，这个主窗口单独加载和编辑文档。这种情况下，多个主窗口属于同一个应用程序，当关闭所有的主窗口的时候，文本编辑器应用程序也就结束了运行。这种方法称为多实例实现编辑器的多文档。
- 使用 `QWorkspace` 作为主窗口的中心部件，在 `QWorkspace` 中打开多个子窗口，每一个子窗口可以单独对文档进行加载和编辑。
- 使用 `QMdiArea` 作为主窗口的中心部件，由 `QMdiArea` 实现对多个文档子窗口的管理（同 `Qworkspace` 的使用有些类似），这个类是 Qt 4.3 新加的。

下面，使用第一种方法实现简单文本编辑器的多文档功能，对于窗口部件 `QWorkspace` 和 `QMdiArea` 的使用将在第 5 章“布局管理”中学习。

在简单文本编辑器的例子中，只有一种情况会出现多文档：当新建一个文档的时候。读者可以进一步完善，比如在打开一个文档的时候，实现多个文档显示。

多实例实现编辑器的多文档比较简单，在前面的基础上，只需要修改两个地方就可以了。

```
// chapter04/designmainwindow/src/mainwindow.cpp.  
void CMainWindow::doNew()  
{  
    CMainWindow* mainWindow = new CMainWindow;  
    mainWindow->setVisible(true);  
}
```

槽函数 `doNew()` 创建一个新的主窗口 `CMainWindow` 对象，并进行显示。

修改函数 `setCurrentFile()`，内容如下。

```
// chapter04/designmainwindow/src/mainwindow.cpp.  
void CMainWindow::setCurrentFile(const QString& fileName)  
{  
    curFile = QFileInfo(fileName).canonicalFilePath();  
}
```

```

isUntitled = false;
setWindowTitle(curFile + "[*]");
textEdit->document()->setModified(false);
setWindowModified(false);

QSettings settings("709", "SDI example");
QStringList files = settings.value("recentFiles").toStringList();
files.removeAll(fileName);
files.prepend(fileName);
while (files.size() > MaxRecentFiles)
    files.removeLast();
settings.setValue("recentFiles", files);

foreach (QWidget *widget, QApplication::topLevelWidgets())
{
    CMainWindow *mainWindow = qobject_cast<CMainWindow *>(widget);
    if (mainWindow)
        mainWindow->updateRecentFiles();
}
}

```

setCurrentFile()函数与原来的实现不同的地方,是多了对所有文本编辑器实例中最近文件列表的修改。

函数 QApplication::topLevelWidgets()返回应用程序的顶级窗口部件列表。对于任何一个文本编辑器实例,都要更新它的最近打开文件列表。

现在,重新编译、运行应用程序。

4.8 打印文档

现在还是以 Qt 设计器绘制的主窗口为例,在主窗口的“工具”菜单中添加一个打印菜单命令,并实现打印文档的功能。

在 CMainWindow 类的头文件 mainwindow.h 中的私有区声明打印动作:

```
QAction* actPrint;
```

在头文件 mainwindow.h 中的 private slots 区声明打印槽函数:

```
void doPrint();
```

接下来,修改 CMainWindow 类的实现文件 mainwindow.cpp,实现打印功能。

修改 iniConnect()函数,添加下列代码。

```
actPrint = menu_T->addAction(tr("打印文档"));
connect(actPrint, SIGNAL(triggered()), this, SLOT(doPrint()));
```

在实现文件 mainwindow.cpp 中添加 doPrint()槽的实现,如下。

```

void CMainWindow::doPrint()
{
    QTextDocument *document = textEdit->document();
    QPrinter printer;

```



```
QPrintDialog dlg(&printer, this);  
dlg.setWindowTitle(tr("打印文档"));  
if (dlg.exec() != QDialog::Accepted)  
    return;  
  
document->print(&printer);  
}
```

在槽函数的最后，调用 `QTextDocument::print()` 函数，实现文档的打印。

现在编译运行应用程序，执行打印功能。如果没有打印机的话，可以在“打印文档”对话框中选择“打印到文件”，将文档打印到一个 PDF 文件，如图 4-25 所示。

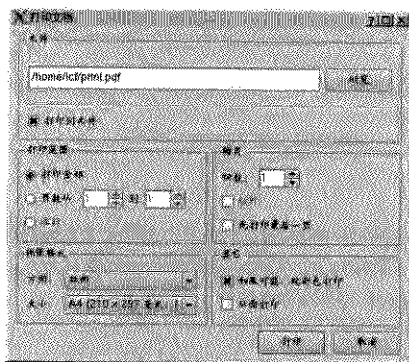


图 4-25 打印文档到 PDF 文件

然后可以打开打印的 PDF 文件，查看打印效果。

4.9 小 结

Qt 应用程序的主窗口是由多个窗口部件/组件构成的框架，包括菜单栏、工具栏、锚接部件、中心部件以及状态栏，这些窗口部件可以单独设计、实现，并组装在一起。这一章，深入介绍了 Qt 应用程序主窗口的这个框架，学习了使用 Qt 设计器绘制和手动编写代码实现一个简单文本编辑器的主窗口，并实现了这个简单的文本编辑器的基本功能。

第5章 布局管理

GUI 用户界面的每一个窗口部件都必须有一个显示位置和大小尺寸,特别是当用户拉伸窗口部件时,窗口部件中的子窗口部件都应该做出适当的变化,以免无法正常显示子窗口部件的内容。在这一章,将学习 Qt 的布局管理器、分裂器以及栈部件和工作空间部件的使用。

5.1 Qt 布局管理器——QLayout

Qt 布局管理器是 Qt 图形用户界面设计的基本内容,在前面的几章中也断断续续讲到了 Qt 的布局管理器的使用,也做了一些简单的应用。本小节将详细介绍 Qt 的布局管理器,详细阐述 Qt 布局管理器的一些特性。

5.1.1 Qt 布局管理器简介

Qt 提供了几种在窗口部件上管理子窗口部件的基本方式。

(1) **绝对位置方式**。这种方式是通过基类 QWidget 提供的 setGeometry()函数来设置子窗口部件的大小及其在父窗口部件中的位置。使用这种方式的缺点有很多,比如用户不能够重定义窗口的大小;再比如,当用户随意的改变窗口的大小时,子窗口部件无法作出相应的改变,子窗口部件的大小以及文本都有可能被截断。此外,使用这种方法必须要程序员反复的计算子窗口部件的大小和位置,并不断地运行、调试和观察 GUI 界面的效果。很显然,使用这种方式管理 GUI 用户界面的外观远远降低了程序员的开发效率,降低了应用程序的质量和适应性。

(2) **手工布局方式**。这种方式也是通过基类 QWidget 提供的 setGeometry()函数来设置子窗口部件的位置和大小。与绝对位置方式不同的是,它是通过重载 QWidget::resizeEvent(QResizeEvent*)函数来实现的。当窗口大小改变的时候,通过重新计算窗口的大小或者变化比例,能够使得子窗口部件的大小和位置作出适应性的改变。但它同样存在一个很严重的问题,就是程序员必须手工计算子窗口部件的大小和位置。因此,使用这种方式进行 GUI 开发,其效率仍然是低下的。

(3) **布局管理器方式**。这种方式是使用 Qt 设计图形化用户界面、组织管理 Qt GUI 窗口部件的最好方法。布局管理器为窗口部件提供了有感知的默认值 (sensible default sizes),可以随着窗口部件大小的变化,对子窗口部件的大小和位置作出适当的调整。

Qt 主要提供了 4 种布局管理器,具体如图 5-1 所示。

- 水平布局管理器 QHBoxLayout,按水平方向组织管理 Qt GUI 窗口部件;
- 垂直布局管理器 QVBoxLayout,按垂直方向组织管理 Qt GUI 窗口部件;
- 网格布局管理器 QGridLayout,按二维网格方式组织管理 Qt GUI 窗口部件;
- 栈布局管理器 QStackedLayout,按照一种类似于栈的方式组织窗口部件,在某一时刻只有一个窗口部件是可见的。Qt 设计器没有提供对该布局管理器的支持,不过提供了一个栈部件 QStackedWidget。因此,使用 Qt 设计器绘制 GUI 界面时,完全可以使用 QStackedWidget 代替



QStackedLayout。

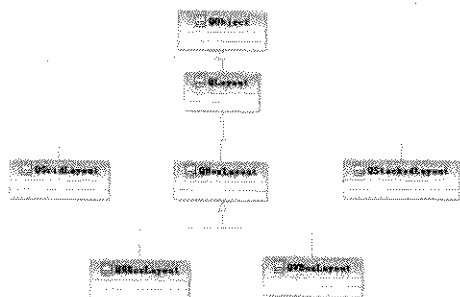


图 5-1 Qt 布局管理器类

Qt 的三种布局管理器的管理模式如图 5-2、图 5-3 和图 5-4 所示。

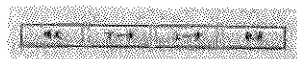


图 5-2 QHBoxLayout

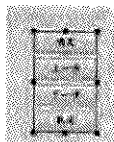


图 5-3 QVBoxLayout

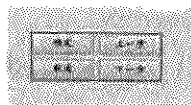


图 5-4 QGridLayout

Qt 布局管理器可以有效地管理窗口中的子窗口部件，同时子窗口部件的布局也受到其本身的大小策略（size policies）及其最大值/最小值的影响。

窗口部件的大小策略告诉布局管理器该窗口部件自身应该如何被拉伸或者压缩的。大小策略包含水平方向的大小策略和垂直方向的大小策略两个方面。每个方向分别有 7 个策略值（其类型为枚举类型 `QSizePolicy::Policy`），它们是：

- `QSizePolicy::Fixed`，窗口部件既不能被拉伸也不能被压缩，它总是保持在大小提示（`sizeHint`）的尺寸。窗口部件的大小范围是 `sizeHint()~sizeHint()`。
- `QSizePolicy::Minimum`，窗口部件的大小提示是它的最小尺寸，即窗口部件的大小不能够被压缩的比大小提示更小。但窗口部件可以被拉伸来填充尽可能多的空间。窗口部件的大小范围是：`sizeHint()~无穷大`。
- `QSizePolicy::Maximum`，窗口部件的大小提示是它的最大尺寸，即窗口部件的大小不能够被拉伸的比大小提示更大，但窗口部件可以被压缩。窗口部件的大小范围是：`0~sizeHint()`。
- `QSizePolicy::Preferred`，窗口部件的大小提示是窗口部件最合适的尺寸；但是如果需要，窗口部件还是能够被拉伸和被压缩的。窗口部件的大小范围是：`0~无穷大`。
- `QSizePolicy::Expanding`，窗口部件可以被拉伸也可以被压缩，但它更倾向于被拉伸。窗口部件的大小范围是：`0~无穷大`。
- `QSizePolicy::MinimumExpanding`，窗口部件的大小提示是它的最小尺寸；然而，它也倾向于被拉伸来填充尽可能多的空间。它相当于 `QSizePolicy::Minimum` 和 `QSizePolicy::Expanding` 的组合。窗口部件的大小范围是：`sizeHint()~无穷大`。

- `QSizePolicy::Ignored`。窗口部件忽略它的大小提示，并且它尽可能的伸展到可用的空间。窗口部件的大小范围是：0~无穷大。

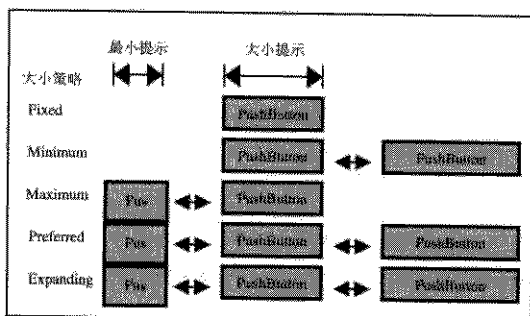


图 5-5 窗口部件大小策略的含义

在介绍 Qt 窗口部件的大小策略的时候，提到一个很重要的概念：大小提示。大小提示是进行 Qt GUI 编程的时候，Qt 推荐的一个窗口部件的大小。当 Qt GUI 窗口部件在进行初始化的时候，将通过虚函数 `QWidget::sizeHint()` 获得该窗口部件的大小提示：

- 如果该窗口部件没有布局管理器，那么该函数将返回一个无效的值；
- 如果该窗口部件具有布局管理器，那么该函数返回一个布局管理器认为比较合适的大小尺寸。

值得注意的是，程序中使用 Qt 设计器绘制的 GUI 界面的时候，顶层窗口部件的大小提示 `sizeHint()` 不再起作用，即调用 `show()`（或 `setVisible(true)`）显示顶层窗口部件的时候，`QWidget::sizeHint()` 函数不再被调用。这就决定了顶层窗口部件的大小策略变化的参照已不再是窗口部件的大小提示，而是窗口部件在 Qt 设计器中的实际大小（准确地讲，应该取设计器中顶层窗口部件的尺寸和顶层窗口部件 `mimimumSizeHint()` 中的较大尺寸），从而达到 Qt 设计器绘制 GUI 用户界面的所见即所得（顶层窗口部件的子窗口部件因为受到其父窗口部件的布局管理器的管理，尺寸会根据各自的大小策略随父窗口部件的变化而变化。这也是在 Qt 设计器中嵌套使用布局管理器的原因，即通过布局管理器一层层的管理，保证 Qt 绘制的 GUI 界面就是应用程序实际运行的用户界面）。

根据 Qt 设计器绘制的 GUI 界面生成的 C++ 头文件中，函数 `setupUi()` 会调用函数 `QWidget::mimimumSizeHint()`，结合 Qt 设计器中顶层窗口部件的实际绘制尺寸，完成顶层窗口部件的大小设置。函数 `QWidget::mimimumSizeHint()` 的返回值遵循下列规则：

- 如果该窗口部件没有布局管理器，`QWidget::mimimumSizeHint()` 函数返回一个无效的值；
- 如果该窗口部件具有布局管理器，`QWidget::mimimumSizeHint()` 函数返回布局管理器认为适合的一个大小尺寸。

最小提示（minimum size hint）是 Qt 为窗口部件推荐的最小尺寸，它的使用情况是（在最小提示有效的情况下）：

如果需要绘制的窗口部件的大小尺寸（包括长宽两个方面）小于其最小提示（这在 Qt 设计器中往往表现有些被压缩的看不到它的内容），并且该窗口部件的最小提示在最大尺寸和最小尺寸允许的范围内，那么该窗口部件显示后的大小尺寸将是最小提示的值。

然而，当多个窗口部件用布局管理器组合在一起的时候，窗口部件的大小及它们是否能够伸展或者缩短是要受到其他窗口部件的大小策略影响的。有下列几种情况：

- 相同大小策略的窗口部件被布局管理器组合在一起。在这种情况下，除了窗口部件不能超出它的大小范围外，不同的窗口部件可以按自己的伸缩因子在其允许的范围内自由地伸缩。
- `QSizePolicy::Fixed` 和任何其他的大小策略组合在一起。具有 `QSizePolicy::Fixed` 大小策略的窗口部件其大小总是不变的，即保持在 `sizeHint()` 大小，而其他的窗口部件可以在允许的范围内自由伸缩。
- `QSizePolicy::Preferred` 和 `QSizePolicy::Expanding`（或者 `QSizePolicy::MinimumExpanding`）组合在一起，具有 `QSizePolicy::Preferred` 大小策略的窗口部件其大小是不变化的，即它认为大小提示 `sizeHint()` 是最适合它的，而其他的窗口部件大小可以在其允许的范围内自由伸缩。
- `QSizePolicy::Ignored` 和其他大小策略（`QSizePolicy::Fixed` 策略除外）组合在一起的时候，不同的窗口部件在各自允许的范围内自由伸缩。
- `QSizePolicy::Preferred`、`QSizePolicy::Minimum` 和 `QSizePolicy::Maximum` 组合在一起的时候，各窗口部件在各自允许的范围内可以自由伸缩。

当多个窗口部件被布局管理器组合在一起的时候，它们都可以在其范围内自由伸缩的情况下，它们伸缩的多少也是互相影响的，这通过设置窗口部件的伸缩因子来实现的，即设置窗口部件大小策略的属性 `QSizePolicy::horizontalStretch` 和 `QSizePolicy::verticalStretch` 的值来实现的。默认情况下，被布局管理器组合在一起的窗口部件的伸缩因子是相等的，都为 0。此时，在所有的窗口部件都没有超出各自的大小范围允许的情况下，窗口部件的大小始终是相等的。

假设，窗口部件 `widget1` 和 `widget2` 被一个布局管理器 `QHBoxLayout` 组合在一起，如图 5-6 所示，并且 `widget1` 的 `QSizePolicy::horizontalStretch` 设置为“1”，而 `widget2` 的 `QSizePolicy::horizontalStretch` 设置为“2”，并且它们的水平大小策略都为 `QSizePolicy::Preferred`，那么窗口部件 `widget1` 和 `widget2` 的大小总是保持在 1:2 的关系，而无论布局管理器在水平方向上伸缩到何种程度。垂直方向的大小关系与它们的大小策略也有类似的关系。



图 5-6 伸缩因子对大小尺寸的影响

此外，主窗口部件（main widget）的大小还要受到其布局管理器的 `sizeConstraint` 属性的约束：

- `QLayout::setDefaultConstraint`，主窗口部件的最小尺寸设置为 `minimumSize()`，除非该窗口部件已经有一个最小尺寸；
- `QLayout::setFixedSize`，主窗口部件的尺寸设置为 `sizeHint()`，并且不允许改变该窗口部件的尺寸；
- `QLayout::setMinimumSize`，主窗口部件的最小尺寸设置为 `minimumSize()`，并且该窗口部件不能够变得更小；
- `QLayout::setMaximumSize`，主窗口部件的最大尺寸设置为 `maximumSize()`，并且该窗口部件不能够变得更大；
- `QLayout::setMinAndMaxSize`，主窗口部件的最小尺寸设置为 `minimumSize()`、最大尺寸设置为 `maximumSize()`；
- `QLayout::setNoConstraint`，主窗口部件的大小不会受到约束。

所谓主窗口部件，就是能够放置一个布局管理器管理其所有子窗口部件的窗口部件。例如，放置了一个布局管理器的“个人信息”`QGroupBox`（它使用一个网格布局管理器管理它内部的所有子窗口部件）。顶层窗口是一个主窗口部件，如图 5-7 所示。

可以通过 `QWidget::layout()` 获取主窗口部件的布局管理器，然后可以调用 `QLayout::setSizeConstraint()` 函数设置该布局管理器的 `sizeConstraint` 属性（将在下面的例子中演示这些属性）。

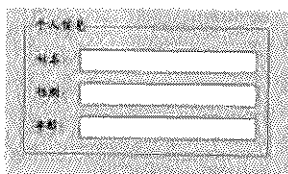


图 5-7 一个主窗口部件的例子

Qt 的布局管理器是 Qt GUI 用户界面设计的一个重要部分。在 Qt 设计器中以及在代码中添加窗口部件、排列组织窗口部件的时候, 总要使用到 Qt 的布局管理器。当多个窗口部件被一个布局管理器组合在一起, 以及多个布局管理器嵌套管理窗口部件的时候, 出现的情况可能还要复杂。在此, 笔者也只是归纳了几种简单使用 Qt 布局管理器的情况, 希望给读者以提示。

Qt 窗口部件的尺寸受到各种各样的策略的影响, 看起来比较复杂。上面的介绍也是比较枯燥的, 下面用一个较简单的例子演示一下布局管理器的使用。

5.1.2 布局管理器及窗口部件大小策略的应用

刚才简述了 Qt 布局管理器的基本原理, 介绍了窗口部件的大小策略对窗口部件伸缩的影响。下面通过一个例子来演示它们的使用。

在学习对话框应用的时候, 曾看到过可以改变形状的 QMessageBox 的一个例子, 如图 5-8 所示。

此处将实现一个类似效果的对话框。为了简单起见, 下面的例子中, 仅仅控制窗口部件的水平方向的策略、伸缩比例等。

在 Qt 设计器绘制 GUI 图形用户界面 (保存为 “layoutdlg.ui”), 如图 5-9 所示。窗口部件属性如表 5-1 所示。

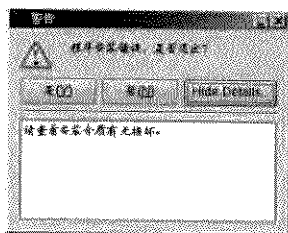


图 5-8 形状可变的对话框

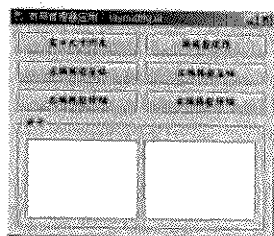


图 5-9 布局管理器应用的 ui 界面

表 5-1 窗口部件属性 (从上到下、从左到右, 缩进表征窗口部件的父子关系)

objectName	text/windowTitle	布局管理器	窗口部件类
LayoutDlg	布局管理器应用	网络布局管理器	QDialog
constraintPushBtn	窗口尺寸约束		QPushButton
showPushBtn	编辑器选项		QPushButton
lpolicyPushBtn	左编辑框策略		QPushButton
rpolicyPushBtn	右编辑框策略		QPushButton
lstretchPushBtn	左编辑框伸缩		QPushButton
rstretchPushBtn	右编辑框伸缩		QPushButton
groupBox	演示	水平布局管理器	QGroupBox
lTextEdit	——		QTextEdit
rTextEdit	——		QTextEdit

下面, 首先来看一下类 CLayoutDlg 的头文件 layoutdlg.h, 如下所示。



```
// chapter05/layout/src/layoutdlg.h
#ifndef _LAYOUTDLG_H_
#define _LAYOUTDLG_H_

#include "ui_layoutdlg.h"
class CLayoutDlg : public QDialog,
                  public Ui_LayoutDlg
{
    Q_OBJECT
public:
    CLayoutDlg(QWidget* = 0);
private:
    void    init();

    QAction*    actDefaultConstraint;
    QAction*    actFixedSize;
    QAction*    actMinimumSize;
    QAction*    actMaximumSize;
    QAction*    actMinAndMaxSize;
    QAction*    actNoConstraint;
```

声明了 6 个动作指针变量，分别对应 QLayout 的 6 个大小约束，用来设置顶层窗口部件的布局管理器的大小约束。

```
QAction*    actLFixed;
QAction*    actLMinimum;
QAction*    actLMaximum;
QAction*    actLPreferred;
QAction*    actLExpanding;
QAction*    actLMinimumExpanding;
QAction*    actLIgnored;
```

声明了 7 个动作指针变量，分别对应 QSizePolicy 的 7 个策略，用于设置 GUI 界面左侧编辑框 QTextEdit 对象 lTextEdit 的水平方向的策略。

```
QAction*    actLStretch0;
QAction*    actLStretch1;
QAction*    actLStretch2;
QAction*    actLStretch4;
```

声明了 4 个动作指针变量，分别对应 QSizePolicy 的 4 个伸缩值，用于设置 GUI 界面左侧编辑框 QTextEdit 对象 lTextEdit 的水平方向的伸缩值。

接下来，继续声明设置 GUI 用户界面的右侧文本编辑框 QTextEdit 对象 rTextEdit 的策略和伸缩值的动作。

```
QAction*    actRFixed;
QAction*    actRMinimum;
QAction*    actRMaximum;
QAction*    actRPreferred;
```

```

    QAction*    actRExpanding;
    QAction*    actRMinimumExpanding;
    QAction*    actRIgnored;
    QAction*    actRStretch0;
    QAction*    actRStretch1;
    QAction*    actRStretch2;
    QAction*    actRStretch4;

private slots:
    void doActConstraint();
    void doActLPolicy();
    void doActRPolicy();
    void doActLStretch();
    void doActRStretch();
};
#endif

```

声明的 5 个槽函数分别响应设置布局管理器的约束、设置左编辑框的策略、设置右编辑框的策略、设置左编辑框的伸缩值和设置右编辑框的伸缩值。

现在，看一下类 CLayoutDlg 的实现文件 layoutdlg.cpp 的内容。

```

// chapter05/layout/src/layoutdlg.cpp
#include <QtGui>
#include "layoutdlg.h"

CLayoutDlg::CLayoutDlg(QWidget* parent)
    : QDialog(parent)
{
    setupUi(this);
    init();
}

```

在构造函数中，调用 init() 初始化函数，初始化 GUI 用户界面的状态。

```

void CLayoutDlg::init()
{
    showPushBtn->setCheckable(true);
    showPushBtn->setChecked(true);
    groupBox->setVisible(showPushBtn->isChecked());
    connect(showPushBtn, SIGNAL(toggled(bool)),
            groupBox, SLOT(setVisible(bool)));
}

```

函数 QPushButton::setCheckable() 设置按钮 showPushBtn 是否具有可选中的 (checked) 状态。在此，设置 showPushBtn 是可选中的。QPushButton::setChecked() 设置按钮的可选中状态为 true (此时，按钮表现为凹陷的)。最后设置“演示”QGroupBox 对象 groupBox 的显隐，并关联信号和槽。

```

QMenu* constraintMenu = new QMenu(constraintPushBtn);
actDefaultConstraint = constraintMenu->addAction(
    tr("QLayout::SetDefaultConstraint"));
actFixedSize = constraintMenu->addAction(
    tr("QLayout::SetFixedSize"));

```



```
actMinimumSize =
    constraintMenu->addAction(tr("QLayout::SetMinimumSize"));
actMaximumSize =
    constraintMenu->addAction(tr("QLayout::SetMaximumSize"));
actMinAndMaxSize =
    constraintMenu->addAction(tr("QLayout::SetMinAndMaxSize"));
actNoConstraint = constraintMenu->addAction(
    tr("QLayout::SetNoConstraint"));
connect(actDefaultConstraint, SIGNAL(triggered()),
    this, SLOT(doActConstraint()));
connect(actFixedSize, SIGNAL(triggered()),
    this, SLOT(doActConstraint()));
connect(actMinimumSize, SIGNAL(triggered()),
    this, SLOT(doActConstraint()));
connect(actMaximumSize, SIGNAL(triggered()),
    this, SLOT(doActConstraint()));
connect(actMinAndMaxSize, SIGNAL(triggered()),
    this, SLOT(doActConstraint()));
connect(actNoConstraint, SIGNAL(triggered()),
    this, SLOT(doActConstraint()));
constraintPushBtn->setMenu(constraintMenu);
```

上面，创建了控制布局管理器的大小约束菜单，并初始化它的所有动作。最后，通过调用函数 `QPushButton::setMenu()` 将该菜单设置到一个 `QPushButton` 对象上。

接下来，使用同样的方法初始化其他的菜单，并将菜单添加到相应的 `QPushButton` 对象上。

```
QMenu* lpolicyMenu = new QMenu(lpolicyPushBtn);
actLFixed = lpolicyMenu->addAction(tr("QSizePolicy::Fixed"));
actLMinimum = lpolicyMenu->addAction(tr("QSizePolicy::Minimum"));
actLMaximum = lpolicyMenu->addAction(tr("QSizePolicy::Maximum"));
actLPreferred = lpolicyMenu->addAction(tr("QSizePolicy::Preferred"));
actLExpanding = lpolicyMenu->addAction(tr("QSizePolicy::Expanding"));
actLMinimumExpanding =
    lpolicyMenu->addAction(tr("QSizePolicy::MinumumExpanding"));
actLIgnored = lpolicyMenu->addAction(tr("QSizePolicy::Ignored"));
connect(actLFixed, SIGNAL(triggered()),
    this, SLOT(doActLPolicy()));
connect(actLMinimum, SIGNAL(triggered()),
    this, SLOT(doActLPolicy()));
connect(actLMaximum, SIGNAL(triggered()),
    this, SLOT(doActLPolicy()));
connect(actLPreferred, SIGNAL(triggered()),
    this, SLOT(doActLPolicy()));
connect(actLExpanding, SIGNAL(triggered()),
    this, SLOT(doActLPolicy()));
connect(actLMinimumExpanding, SIGNAL(triggered()),
    this, SLOT(doActLPolicy()));
connect(actLIgnored, SIGNAL(triggered()),
```



```

        this, SLOT(doActLPolicy()));
lpolicyPushBtn->setMenu(lpolicyMenu);

QMenu* rpolicyMenu = new QMenu(rpolicyPushBtn);
actRFixed =
    rpolicyMenu->addAction(tr("QSizePolicy::Fixed"));
actRMinimum =
    rpolicyMenu->addAction(tr("QSizePolicy::Minimum"));
actRMaximum =
    rpolicyMenu->addAction(tr("QSizePolicy::Maximum"));
actRPreferred =
    rpolicyMenu->addAction(tr("QSizePolicy::Preferred"));
actRExpanding =
    rpolicyMenu->addAction(tr("QSizePolicy::Expanding"));
actRMinimumExpanding =
    lpolicyMenu->addAction(tr("QSizePolicy::MinimumExpanding"));
actRIgnored =
    rpolicyMenu->addAction(tr("QSizePolicy::Ignored"));
connect(actRFixed, SIGNAL(triggered()),
        this, SLOT(doActRPolicy()));
connect(actRMinimum, SIGNAL(triggered()),
        this, SLOT(doActRPolicy()));
connect(actRMaximum, SIGNAL(triggered()),
        this, SLOT(doActRPolicy()));
connect(actRPreferred, SIGNAL(triggered()),
        this, SLOT(doActRPolicy()));
connect(actRExpanding, SIGNAL(triggered()),
        this, SLOT(doActRPolicy()));
connect(actRMinimumExpanding, SIGNAL(triggered()),
        this, SLOT(doActRPolicy()));
connect(actRIgnored, SIGNAL(triggered()),
        this, SLOT(doActRPolicy()));
rpolicyPushBtn->setMenu(rpolicyMenu);

QMenu* lstretchMenu= new QMenu(lstretchPushBtn);
actLStretch0 = lstretchMenu->addAction(tr("伸缩比例 0"));
actLStretch1 = lstretchMenu->addAction(tr("伸缩比例 1"));
actLStretch2 = lstretchMenu->addAction(tr("伸缩比例 2"));
actLStretch4 = lstretchMenu->addAction(tr("伸缩比例 4"));
connect(actLStretch0, SIGNAL(triggered()),
        this, SLOT(doActLStretch()));
connect(actLStretch1, SIGNAL(triggered()),
        this, SLOT(doActLStretch()));
connect(actLStretch2, SIGNAL(triggered()),
        this, SLOT(doActLStretch()));
connect(actLStretch4, SIGNAL(triggered()),
        this, SLOT(doActLStretch()));
lstretchPushBtn->setMenu(lstretchMenu);

```



```
QMenu* rstretchMenu= new QMenu(rstretchPushBtn);
actRStretch0 = rstretchMenu->addAction(tr{"伸缩比例 0"});
actRStretch1 = rstretchMenu->addAction(tr{"伸缩比例 1"});
actRStretch2 = rstretchMenu->addAction(tr{"伸缩比例 2"});
actRStretch4 = rstretchMenu->addAction(tr{"伸缩比例 4"});
connect(actRStretch0, SIGNAL(triggered()),
        this, SLOT(doActRStretch()));
connect(actRStretch1, SIGNAL(triggered()),
        this, SLOT(doActRStretch()));
connect(actRStretch2, SIGNAL(triggered()),
        this, SLOT(doActRStretch()));
connect(actRStretch4, SIGNAL(triggered()),
        this, SLOT(doActRStretch()));
rstretchPushBtn->setMenu(rstretchMenu);
}
```

下面的槽函数 `doActConstraint()` 响应用户选择“窗口大小约束”菜单的动作发出的信号，设置顶层窗口部件的布局管理器的大小约束。

```
void CLayoutDlg::doActConstraint()
{
    QAction* act = qobject_cast<QAction*>(sender());
    if(act == actDefaultConstraint)
        layout()->setSizeConstraint(QLayout::SetDefaultConstraint);
    else if(act == actFixedSize)
        layout()->setSizeConstraint(QLayout::SetFixedSize);
    else if(act == actMinimumSize)
        layout()->setSizeConstraint(QLayout::SetMinimumSize);
    else if(act == actMaximumSize)
        layout()->setSizeConstraint(QLayout::SetMaximumSize);
    else if(act == actNoConstraint)
        layout()->setSizeConstraint(QLayout::SetNoConstraint);
    constraintPushBtn->setText(act->text());
}
```

函数 `QWidget::layout()` 获取安装在当前窗口部件（此处为顶层窗口）的布局管理器。

函数 `QLayout::setSizeConstraint()` 设置布局管理器的大小约束。

最后，设置 `QPushButton` 对象的显示文本为选择的大小约束。

```
void CLayoutDlg::doActLPolicy()
{
    QAction* act = qobject_cast<QAction*>(sender());
    QSizePolicy policy = lTextEdit->sizePolicy();
    if(act == actLFixed)
        policy.setHorizontalPolicy(QSizePolicy::Fixed);
    else if(act == actLMinimum)
        policy.setHorizontalPolicy(QSizePolicy::Minimum);
    else if(act == actLMaximum)
        policy.setHorizontalPolicy(QSizePolicy::Maximum);
}
```

```

else if(act == actLPreferred)
    policy.setHorizontalPolicy(QSizePolicy::Preferred);
else if(act == actLExpanding)
    policy.setHorizontalPolicy(QSizePolicy::Expanding);
else if(act == actLMinimumExpanding)
    policy.setHorizontalPolicy(QSizePolicy::MinimumExpanding);
else if(act == actLIgnored)
    policy.setHorizontalPolicy(QSizePolicy::Ignored);
lTextEdit->setSizePolicy(policy);
lpolicyPushBtn->setText(act->text());
}

```

函数 `doActLPolicy()` 设置左侧文本编辑框 `QTextEdit` 对象水平方向的策略。

函数 `QTextEdit::sizePolicy()` 获取文本编辑框的大小策略, 该函数继承自 `QWidget` 类。

函数 `QSizePolicy::setHorizontalPolicy()` 设置大小策略对象 (`QSizePolicy` 对象) 的水平方向的策略。

最后, 函数 `QTextEdit::setSizePolicy()` 为文本编辑框设置新的大小策略; 设置相应按钮的显示文本。

```

void CLayoutDlg::doActRPolicy()
{
    QAction* act = qobject_cast<QAction*>(sender());
    QSizePolicy policy = rTextEdit->sizePolicy();
    if(act == actRFixed)
        policy.setHorizontalPolicy(QSizePolicy::Fixed);
    else if(act == actRMinimum)
        policy.setHorizontalPolicy(QSizePolicy::Minimum);
    else if(act == actRMaximum)
        policy.setHorizontalPolicy(QSizePolicy::Maximum);
    else if(act == actRPreferred)
        policy.setHorizontalPolicy(QSizePolicy::Preferred);
    else if(act == actRExpanding)
        policy.setHorizontalPolicy(QSizePolicy::Expanding);
    else if(act == actRMinimumExpanding)
        policy.setHorizontalPolicy(QSizePolicy::MinimumExpanding);
    else if(act == actRIgnored)
        policy.setHorizontalPolicy(QSizePolicy::Ignored);
    rTextEdit->setSizePolicy(policy);
    rpolicyPushBtn->setText(act->text());
}

```

函数 `doActRPolicy()` 设置右侧文本编辑框 `rTextEdit` 对象的水平方向的策略。

```

void CLayoutDlg::doActLStretch()
{
    QAction* act = qobject_cast<QAction*>(sender());
    QSizePolicy policy = rTextEdit->sizePolicy();
    if(act == actLStretch0)
        policy.setHorizontalStretch(0);
    else if(act == actLStretch1)
        policy.setHorizontalStretch(1);
    else if(act == actLStretch2)

```



```

        policy.setHorizontalStretch(2);
    else if(act == actLStretch4)
        policy.setHorizontalStretch(4);
    lTextEdit->setSizePolicy(policy);
    lstretchPushBtn->setText(act->text());
}

```

函数 doActLStretch() 设置左侧文本编辑框 lTextEdit 大小策略的水平伸缩值。

函数 QSizePolicy::setHorizontalStretch() 设置大小策略对象的水平伸缩值。

设置 GUI 用户界面右侧文本编辑框大小策略的水平伸缩值与上面类似, 实现代码如下。

```

void CLayoutDlg::doActRStretch()
{
    QAction* act = qobject_cast<QAction*>(sender());
    QSizePolicy policy = rTextEdit->sizePolicy();
    if(act == actRStretch0)
        policy.setHorizontalStretch(0);
    else if(act == actRStretch1)
        policy.setHorizontalStretch(1);
    else if(act == actRStretch2)
        policy.setHorizontalStretch(2);
    else if(act == actRStretch4)
        policy.setHorizontalStretch(4);
    rTextEdit->setSizePolicy(policy);
    rstretchPushBtn->setText(act->text());
}

```

现在, 建立新的 KDevelop 工程 layout, 创建主程序并将刚才建立的 ui 文件和类 CLayoutDlg 的定义文件和实现文件加入到工程中。最后编译运行应用程序, 效果如图 5-10 所示。

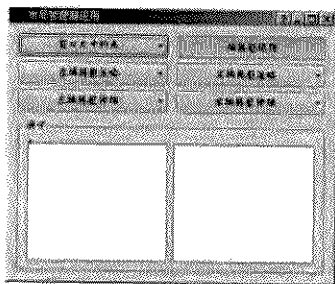


图 5-10 布局管理器应用的运行界面

在这个例子中, 当管理器的策略设置为 QLayout::SetFixedSize 时, 就是一个可变形的对话框。读者可以试着组合这些策略, 体会一下 Qt 布局管理的一些特性。

5.2 分裂器部件 QSplitter

Qt 的分裂器 QSplitter 可以包含其他窗口部件并且通过分裂柄分隔其中的子窗口部件。用户可以通过

过拖动分裂柄改变分裂器的子窗口部件的大小。对用户来讲,使用分裂器管理 GUI 窗口部件比布局管理器更方便灵活,用户可以自由的改变窗口部件的大小。

Qt 设计器提供了对分裂器的支持。在 Qt 设计器中,在顶层窗口部件中,加入两个 QTextEdit 窗口部件 textEdit1 和 textEdit2,下面将这两个窗口部件水平加入到一个分裂器中:

01 同时选中两个 QTextEdit 窗口部件: textEdit1 和 textEdit2。

02 单击工具按钮“Lay Out Horizontal in Splitter”,或者在选中的窗口部件上单击鼠标右键,选择弹出的上下文菜单“Lay Out”|“Lay Out Horizontally in Splitter”(如图 5-11 所示)。加入到一个分裂器的显示效果如图 5-12 所示。

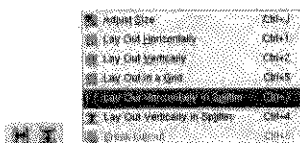


图 5-11 Qt 设计器中的分裂器按钮和菜单

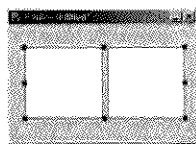


图 5-12 两个窗口部件水平加入到一个分裂器

在 Qt 设计器的对象监视器中,可以看到有一个名字为 splitter 的分裂器 QSplitter 对象,在程序中可以引用该对象完成对分裂器的操作。通过同样的方法也可以将窗口部件垂直放置在一个分裂器中。此外还可以进行分裂器的嵌套,比如将两个水平放置 QTextEdit 窗口部件的分裂器垂直排列到一个新的分裂器,如图 5-13 所示。

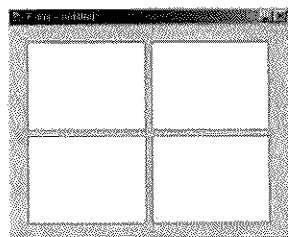


图 5-13 分裂器的嵌套

使用代码实现上述功能也很简单,代码如下。

```
QSplitter* splitter1 = new QSplitter(Qt::Horizontal);
QTextEdit* editor1 = new QTextEdit;
QTextEdit* editor2 = new QTextEdit;
splitter1.addWidget(editor1);
splitter1.addWidget(editor2);

QSplitter* splitter2 = new QSplitter(Qt::Horizontal);
QTextEdit* editor3 = new QTextEdit;
QTextEdit* editor4 = new QTextEdit;
splitter2.addWidget(editor3);
splitter2.addWidget(editor4);
```

```
QSplitter* splitter = new QSplitter(Qt::vertical);
splitter.addWidget(split1);
splitter.addWidget(split2);
```

QSplitter 构造函数的参数指定了分裂器子窗口部件的排列方向:

- Qt::Horizontal, 按水平方向排列;
- Qt::Vertical, 按垂直方向排列。

5.3 栈部件 QStackedWidget

栈部件 QStackedWidget 提供了一个管理窗口部件的栈, 在某一时刻只有一个栈部件的子窗口部件可见。

下面, 在 Qt 设计器中绘制一个具有两页的栈部件的 GUI 界面:

01 在 Qt 设计器中新建一个 QWidget 窗口部件 widget。

02 选择 “Widget Box” 面板 “Item Views (Item-Based)” 中的 “List Widget”, 并将它拖放到 widget 窗口部件左侧, 单击右键, 选择 “Edit items” 菜单, 添加两个选项 “个人设置” 和 “系统设置”, 确定。效果如图 5-14、图 5-15 所示。

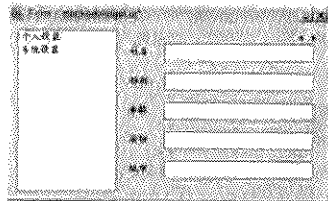


图 5-14 栈部件的配置 “个人设置” 页

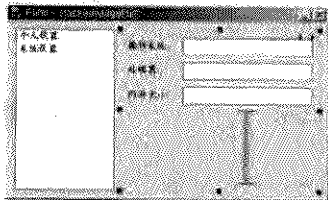


图 5-15 栈部件的系统设置页

03 选择 “Widget Box” 面板 Containers 中的 “Stacked Widget” 窗口部件, 并将它拖放到顶层窗口部件中, 调整栈部件的大小。

04 添加子窗口部件到栈部件, 并将栈部件的当前页放置在一个网格布局管理器中 (如图 5-14 所示)。

05 在栈部件上单击右键, 在弹出的背景菜单中选择 “Insert Page” | “After Current Page”, 插入新的一页。

06 单击栈部件上的 “箭头” 按钮进入下一页, 添加子窗口部件, 并将当前页放置在一个网格布局管理器中 (如图 5-15 所示)。

07 将顶级窗口部件 widget 放置在一个水平布局管理器中, 并设置 listWidget 的 sizePolicy 的 horizontalStretch 值为 “1”, 设置 stackedWidget 的 sizePolicy 的 horizontalStretch 值为 “2”。

现在, 使用 Qt 设计器的信号/槽编辑器为 listWidget 和 stackedWidget 窗口部件关联信号和槽, 操作步骤如下:

01 选择菜单 “Edit” | “Edit Signals/Slots” 或直接按快捷键 “F4”, 进入编辑信号/槽模式。

02 将鼠标移动到 listWidget 窗口部件 (这时该窗口部件高亮显示), 按下鼠标左键的同时将鼠标拖动到 stackedWidget 窗口部件 (此时 stackedWidget 窗口部件也高亮显示, 如图 5-16 所示), 松开鼠标按键。这时候将出现 “Configure Connection” 对话框。

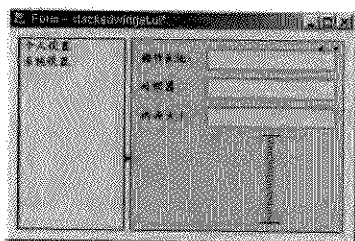


图 5-16 stackedWidget 高亮显示

03 在“Configure Connection”对话框选择相应的信号和槽（QListWidget::currentRowChanged(int)关联到 QStackedWidget::setCurrentIndex(int)），单击“确定”按钮。如图 5-17 所示。

现在，在 Qt 设计器中预览 GUI 界面，试着交替单击列表框的“个人设置”和“系统设置”。显示效果如图 5-18 所示。

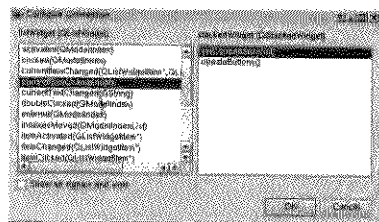


图 5-17 配置 QListWidget 和 stackedWidget 的关联

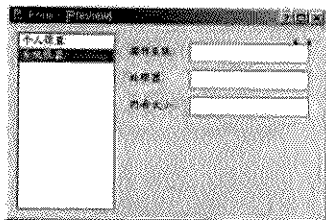


图 5-18 栈部件实例对话框

手写代码也可以实现上述的功能，在此省略。有兴趣的读者可以试着手写代码实现这个同样功能的栈部件。

5.4 工作空间部件 QWorkspace

工作空间部件 QWorkspace 提供了一种多文档界面（Multiple Document Interface, MDI）应用程序的实现方法，它可以包容、管理多个子窗口。在一个这样的 MDI 应用程序中，工作空间部件是作为主窗口的中心部件来使用的，它的子窗口具有和一般的顶层窗口一样的行为模式，比如可以显示、隐藏、最大化和设置窗口标题等。

下面，通过使用工作空间部件来实现第 4 章的简单文本编辑器的多文档功能。

为了保证工作空间部件的子窗口在关闭时能够保存文档的数据，在不使用事件过滤器（事件过滤器将在第 13 章“事件处理”中讲述）的情况下，必须要定义一个继承自 QTextEdit 的自定义窗口部件来实现编辑器的功能，并重写虚函数 closeEvent()。

第 4 章已经详细描述了简单文本编辑器的实现，尽管本节的例子中自定义了一个 CMDITextEdit 类，但一些基本的函数功能还是相同的。在此主要解释它们的不同点。

首先，看一下自定义类 CMDITextEdit 的头文件 mditextedit.h。

```
// chapter05/workspace/src/mditextedit.h.
```



```
#ifndef _MDITEXTEDIT_H_
#define _MDITEXTEDIT_H_

#include <QTextEdit>
class CMDITextEdit : public QTextEdit
{
    Q_OBJECT
public:
    CMDITextEdit(bool , const QString &);
    virtual ~CMDITextEdit();
    bool loadFile(const QString &fileName);
    bool save();
    bool saveAs();
    bool saveFile(const QString &fileName);
    QString fileName();

    static int count;
protected:
    void closeEvent(QCloseEvent *event);
private:
    bool maybeSave();
    void setCurrentFile(const QString &fileName);

    QString curFile;
    bool isUntitled;
    enum(MaxRecentFiles = 9);
signals:
    void updateRecentFiles();
    void counted(int);
private slots:
    void doModified();
};
#endif
```

自定义类 CMDITextEdit 继承自 QTextEdit，实现文本的编辑以及状态的保存。

公有函数 fileName() 获取当前文档的文件名字。

公有静态变量 count 对新建（或打开）的文档进行计数。

信号 updateRecentFiles() 用于打开或第一次保存时，以通知应用程序更新最近文件列表。

信号 counted() 通知应用程序在状态栏显示新的文档计数。

下面看一下它的实现文件 mditextedit.cpp。

```
// chapter05/workspace/src/mditextedit.cpp
#include <QtGui>
#include "mditextedit.h"

int CMDITextEdit::count = 0;
CMDITextEdit::CMDITextEdit(bool untitled, const QString& fileName)
: isUntitled(untitled),
```



```

    curFile(fileName)
{
    setAttribute(Qt::WA_DeleteOnClose);
    setWindowTitle(fileName + ".*");
    connect(document(), SIGNAL(contentsChanged()),
            this, SLOT(doModified()));
}

```

实现文件 `mditextedit.cpp` 中, 必须首先要对静态成员变量 `count` 进行初始化。

类 `CMDITextEdit` 的构造函数有两个形参 `untitled` 和 `fileName`, 第一个参数告诉新建的 `CMDITextEdit` 对象新文档是否保存过: 参数 `fileName` 指定了新建文档的名字。

函数 `setAttribute()` 继承自 `QWidget` 类, 设置窗口部件对象的属性为 `Qt::WA_DeleteOnClose`, 即当窗口部件接受关闭事件的时候, 销毁掉该窗口部件对象。

```

CMDITextEdit::~CMDITextEdit()
{
    emit counted(--count);
}

```

在析构函数中, 对文档计数变量进行减 1 操作, 并发送信号 `counted()`, 以通知应用程序在主窗口的状态栏上显示打开的文档总数。

```

QString CMDITextEdit::fileName()
{
    return curFile;
}

```

函数 `fileName()` 返回当前文档的文件名字。当用户打开文件时, 将该函数返回的文件名同用户打开的文件名进行比较, 如果相同则不再需要打开该文件。

```

bool CMDITextEdit::save()
{
    if (isUntitled)
    {
        saveAs();
    }
    else
    {
        saveFile(curFile);
        document()->setModified(false);
        setWindowModified(false);
    }
}

```

函数 `save()` 完成文档的保存。如果文档从来没有保存过, 执行另存为操作; 否则直接保存并设置底层文档以及文本编辑框窗口部件的状态。

```

bool CMDITextEdit::saveAs()
{
    QString fileName =

```



```
QFileDialog::getSaveFileName(this, tr("另存为"), curFile);
if (!fileName.isEmpty())
{
    saveFile(fileName);
    setCurrentFile(fileName);
}
}
```

函数 `saveAs()` 完成文档的另存为操作。

```
bool CMDITextEdit::loadFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly | QFile::Text))
    {
        QMessageBox::warning(this, tr("读取文件"),
                               tr("无法读取文件 %1:\n%2.")
                               .arg(fileName)
                               .arg(file.errorString()));
        return false;
    }

    QTextStream in(&file);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    setText(in.readAll());
    QApplication::restoreOverrideCursor();
    setCurrentFile(fileName);

    return true;
}
```

函数 `loadFile()` 完成文件的加载，并设置当前文本编辑框窗口的状态。

```
bool CMDITextEdit::saveFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::WriteOnly | QFile::Text))
    {
        QMessageBox::warning(this,
                               tr("保存文件"),
                               tr("无法保存文件 %1:\n%2.")
                               .arg(fileName)
                               .arg(file.errorString()));
        return false;
    }

    QTextStream out(&file);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    out << toPlainText();
    QApplication::restoreOverrideCursor();
}
```

```

    return true;
}

```

函数 `saveFile()` 实现文本编辑框窗口的文档保存。

```

void CMDITextEdit::closeEvent(QCloseEvent *event)
{
    if (maybeSave())
        event->accept();
    else
        event->ignore();
}

```

重写函数 `closeEvent()` 接收窗口部件的“关闭”事件，如果文档还没有保存，则提示用户保存文件；否则直接关闭该文档的文本编辑框窗口。

```

void CMDITextEdit::doModified()
{
    setWindowModified(document()->isModified());
}

```

槽函数 `doModified()` 接收文本编辑框窗口的 `textChanged()` 信号，设置窗口的编辑状态。

```

bool CMDITextEdit::maybeSave()
{
    if(document()->isModified())
    {
        QMessageBox box;
        box.setWindowTitle(tr("警告"));
        box.setIcon(QMessageBox::Warning);
        box.setText(tr("文档没有保存, 是否保存? "));
        box.setStandardButtons(QMessageBox::Yes
                               | QMessageBox::No
                               | QMessageBox::Cancel);
        switch(box.exec())
        {
            case QMessageBox::Yes :
                save();
                return true;
            case QMessageBox::No :
                return true;
            case QMessageBox::Cancel :
                return false;
        }
    }
    else
        return true;
}

```

一般当用户退出文本编辑器应用程序或关闭某个文本编辑框窗口时，调用函数 `maybeSave()`，提示用



户是否保存文档:

- 用户选择是, 则保存文档, 返回“true”(通知调用函数接受“关闭”事件);
- 用户选择否, 不保存文档, 返回“true”(通知调用函数接受“关闭”事件);
- 用户选择取消, 不保存文件, 返回“false”(通知调用函数忽略“关闭”事件)。

```
void CMDITextEdit::setCurrentFile(const QString &fileName)
{
    curFile = QFileInfo(fileName).canonicalFilePath();
    isUntitled = false;
    setWindowTitle(curFile + "[*]");
    document()->setModified(false);
    setWindowModified(false);

    QSettings settings("709", "MDI example");
    QStringList files = settings.value("recentFiles").toStringList();
    files.removeAll(fileName);
    files.prepend(fileName);
    while (files.size() > MaxRecentFiles)
        files.removeLast();
    settings.setValue("recentFiles", files);

    emit updateRecentFiles();
}
```

函数 `setCurrentFile()` 完成最近文件列表的更新和保存, 并发送信号 `updateRecentFiles()` 通知应用程序更新文件菜单中的最近文件列表的内容。

主窗口类 `CMainWindow` 和前面第 4 章简单文本编辑器的主窗口基本相同, 不同的是添加了工作空间部件作为主窗口的中心部件, 以实现文本编辑器的多文档功能。

主窗口类 `CMainWindow` 的定义文件 `mainwindow.h` 内容如下。

```
// chapter05/workspace/src/mainwindow.h
#ifndef _MAINWINDOW_H_
#define _MAINWINDOW_H_

#include "ui_mainwindow.h"
class QLabel;
class QWorkspace;
class QSignalMapper;
class CMDITextEdit;

class CMainWindow : public QMainWindow,
                   public Ui::MainWindow
{
    Q_OBJECT
public:
    CMainWindow(QWidget* = 0);

private:
```

```

QDockWidget*   dockWidget;
QLabel*        label1;
QLabel*        label2;
enum{MaxRecentFiles = 9};
QAction*       recentFileActs[MaxRecentFiles];
QAction*       separatorAct;
QWorkspace*    workspace;

void           iniDockWidget();
void           iniStatusBar();
void           iniConnect();
CMDITextEdit* activeWindow();
CMDITextEdit* createMDITextEdit(bool, const QString&);

```

在类 CMainWindow 的私有区，新增了必要的公共成员变量和两个成员函数。

成员变量 workspace 声明了一个工作空间部件 QWorkspace 的对象指针。

成员函数 activeWindow() 获取当前应用程序的活动文本编辑框窗口；createMDITextEdit() 窗口创建新的文本编辑框窗口。

```

private slots:
    void doNew();
    void doOpen();
    void doClose();
    void doSave();
    void doASave();
    void doQuit();
    void doUndo();
    void doCut();
    void doCopy();
    void doPaste();
    void doAll();
    void doFind();
    void openRecentFile();
    void updateRecentFiles();
    void updateMenu();
    void showCount(int);
};
#endif

```

在类 CMainWindow 的私有槽函数声明区，去掉了一些不再使用的槽函数，新加了两个槽函数：updateMenu() 在多文档切换的时候更新菜单的状态；showCount(int) 在主窗口的状态栏显示文档的总个数。

下面是主窗口类 CMainWindow 的实现文件 mainwindow.cpp 的内容。

```

// chapter05/workspace/src/mainwindow.cpp
#include <QtGui>
#include <QDebug>
#include <QtCore>

#include "mainwindow.h"

```



```
#include "findfileform.h"
#include "mditextedit.h"

CMainWindow::CMainWindow(QWidget* parent)
: QMainWindow(parent)
{
    setupUi(this);

    workspace = new QWorkspace;
    setCentralWidget(workspace);
    connect(workspace, SIGNAL(windowActivated(QWidget *)),
            this, SLOT(updateMenu()));

    iniDockWidget();
    iniStatusBar();
    iniConnect();
    updateRecentFiles();
    updateMenu();

    showMaximized();
    showCount(0);
}
```

构造函数完成应用程序主窗口及其状态的初始化。

主窗口的中心部件初始化为一个工作空间部件。当用户激活工作空间部件的某个子窗口时，工作空间部件将会发出信号 `QWorkspace::windowActivated()`，信号的参数是被激活的子窗口指针。此处，将该信号关联到槽函数 `updateMenu()`，当新的子窗口被激活时，完成菜单的更新。

```
void CMainWindow::iniDockWidget()
{
    CFindFileForm* findFileForm = new CFindFileForm;
    dockWidget = new QDockWidget(tr("查找文件"), this);
    dockWidget->setAllowedAreas(Qt::RightDockWidgetArea);
    dockWidget->setFeatures(QDockWidget::AllDockWidgetFeatures);
    dockWidget->setFloating(false);
    dockWidget->setWidget(findFileForm);
    dockWidget->setVisible(false);

    addDockWidget(Qt::RightDockWidgetArea, dockWidget);
}
```

函数 `iniDockWidget()` 完成对锚接部件的初始化，该函数的实现基本上没有变化。

```
void CMainWindow::iniStatusBar()
{
    QStatusBar* bar = statusBar();
    label1 = new QLabel;
    label1->setMinimumSize(200, 25);
}
```

```

label1->setFrameShadow(QFrame::Sunken);
label1->setFrameShape(QFrame::WinPanel);
label2 = new QLabel;
label2->setMinimumSize(200, 25);
label2->setFrameShadow(QFrame::Sunken);
label2->setFrameShape(QFrame::WinPanel);
bar->addWidget(label1);
bar->addWidget(label2);
}

```

函数 `iniStatusBar()` 完成对状态栏窗口部件的初始化。

```

void CMainWindow::iniConnect()
{
    connect(actNew, SIGNAL(triggered()), this, SLOT(doNew()));
    connect(actOpen, SIGNAL(triggered()), this, SLOT(doOpen()));
    connect(actClose, SIGNAL(triggered()), this, SLOT(doClose()));
    connect(actSave, SIGNAL(triggered()), this, SLOT(doSave()));
    connect(actASave, SIGNAL(triggered()), this, SLOT(doASave()));
    connect(actQuit, SIGNAL(triggered()), this, SLOT(doQuit()));
    connect(actUndo, SIGNAL(triggered()), this, SLOT(doUndo()));
    connect(actCut, SIGNAL(triggered()), this, SLOT(doCut()));
    connect(actCopy, SIGNAL(triggered()), this, SLOT(doCopy()));
    connect(actPaste, SIGNAL(triggered()), this, SLOT(doPaste()));
    connect(actAll, SIGNAL(triggered()), this, SLOT(doAll()));
    connect(actFind, SIGNAL(triggered()), this, SLOT(doFind()));

    separatorAct = menu_F->insertSeparator(actQuit);
    separatorAct->setVisible(false);
    for (int i = 0; i < 9; ++i)
    {
        recentFileActs[i] = new QAction(this);
        menu_F->insertAction(separatorAct, recentFileActs[i]);
        recentFileActs[i]->setVisible(false);
        connect(recentFileActs[i], SIGNAL(triggered()),
            this, SLOT(openRecentFile()));
    }
}

```

函数 `iniConnect()` 完成信号和槽的关联。

```

void CMainWindow::doFind()
{
    dockWidget->show();
    dockWidget->setFloating(false);
}

```

槽函数 `doFind()` 控制锚接部件的显隐。

```

void CMainWindow::doNew()
{

```



```
static int sequenceNum = 1;
CMDITextEdit* textEdit =
    createMDITextEdit(true, tr("untitled%i").arg(sequenceNum++));
textEdit->setVisible(true);
}
```

槽函数 doNew()响应用户的“新建”操作，创建文本编辑框窗口并进行显示。

局部静态变量 sequenceNum 记录文档的序列号，作为新建文档的标题的一部分。注意，局部静态变量的初始化是在第一次使用的时候，当再次使用的时候不再进行初始化。

```
void CMainWindow::doOpen()
{
    QString fileName = QFileDialog::getOpenFileName(this);
    if (!fileName.isEmpty())
    {
        QWidgetList list = workspace->windowList();
        for(int i=0; i<list.count(); i++)
        {
            CMDITextEdit* textEdit =
                qobject_cast<CMDITextEdit*>(list[i]);
            if(textEdit->fileName() == fileName)
            {
                workspace->setActiveWindow(textEdit);
                return ;
            }
        }

        CMDITextEdit* textEdit
            = createMDITextEdit(false, fileName);
        if (textEdit->loadFile(fileName))
        {
            label2->setText(tr("已经读取"));
        }
        textEdit->setVisible(true);
    }
}
```

槽函数 doOpen()创建一个新的文本编辑框窗口，并在新建的窗口中打开一个文本文档。

函数 QWorkspace::setActiveWindow()设置工作空间部件的活动子窗口。

```
void CMainWindow::doClose()
{
    activeWindow()->close();
}
```

槽函数 doClose()响应用户“关闭”当前文本编辑框窗口的操作，调用文本编辑框窗口的“关闭”槽函数。

```
void CMainWindow::doSave()
{

```



```

    activeWindow()->save();
}

```

槽函数 `doSave()`响应用户的“保存”操作，调用活动文本编辑框窗口的“保存”函数。

```

void CMainWindow::doASave()
{
    activeWindow()->saveAs();
}

```

槽函数 `doSave()`响应用户的“另存为”操作，调用活动文本编辑框窗口的“另存为”函数。

```

void CMainWindow::doQuit()
{
    QWidgetList list = workspace->windowList();
    for(int i=0; i<list.count(); i++)
    {
        CMDITextEdit* textEdit =
            qobject_cast<CMDITextEdit*>(list[i]);
        textEdit->close();
    }
    qApp->quit();
}

```

槽函数 `doQuit()`响应用户的“退出”文本编辑器操作，对当前打开的所有文档进行检查，对还没有保存的文档进行“保存”操作提示（比较完善的做法是，重写 `CMainWindow` 的 `closeEvent()`函数，以同样的方法处理主窗口的“关闭”操作）。

下面几个槽函数完成活动文本编辑框窗口的编辑操作，包括撤消、剪切、复制、粘贴和选择全部。

```

void CMainWindow::doUndo()
{
    activeWindow()->undo();
}

void CMainWindow::doCut()
{
    activeWindow()->cut();
}

void CMainWindow::doCopy()
{
    activeWindow()->copy();
}

void CMainWindow::doPaste()
{
    activeWindow()->paste();
}

void CMainWindow::doAll()

```



```

{
    activeWindow()->selectAll();
}

void CMainWindow::openRecentFile()
{
    QAction *action = qobject_cast<QAction *>(sender());
    if (action)
    {
        QString fileName = action->data().toString();
        CMDITextEdit* textEdit =
            createMDITextEdit(false, fileName);
        textEdit->loadFile(fileName);
        textEdit->setVisible(true);
    }
}

```

槽函数 `openRecentFile()` 完成用户的打开文件菜单中的最近文件列表文件的操作，创建新的文本编辑框窗口并加载指定的文件。

```

void CMainWindow::updateRecentFiles()
{
    QSettings settings("709", "MDI example");
    QStringList files = settings.value("recentFiles").toStringList();

    int numRecentFiles = qMin(files.size(), (int)9);
    for (int i = 0; i < numRecentFiles; ++i)
    {
        QString text = tr("&%1 %2").arg(i + 1).arg(files[i]);
        recentFileActs[i]->setText(text);
        recentFileActs[i]->setData(files[i]);
        recentFileActs[i]->setVisible(true);
    }
    for (int i = numRecentFiles; i < 9; ++i)
        recentFileActs[i]->setVisible(false);

    separatorAct->setVisible(numRecentFiles > 0);
}

```

槽函数 `updateRecentFiles()` 完成应用程序主窗口的文件菜单的最近文件列表的更新，与第 4 章具有相同标签的函数不同的是，此处的 `updateRecentFiles()` 是一个槽，它接收来自 `CMDITextEdit` 对象的信号 `updateRecentFiles()`。

```

CMDITextEdit* CMainWindow::activeWindow ()
{
    CMDITextEdit* textEdit =
        qobject_cast<CMDITextEdit*>(workspace->activeWindow());
    return textEdit;
}

```

函数 `activeWindow()` 获取并返回工作空间部件的当前活动窗口。

函数 `QWorkspace::activeWindow()` 获取工作空间部件的当前活动的子窗口。

```
void CMainWindow::updateMenu()
{
    bool hasChild = (activeWindow() != 0);
    actSave->setEnabled(hasChild);
    actASave->setEnabled(hasChild);
    actPaste->setEnabled(hasChild);
    actClose->setEnabled(hasChild);
    actUndo->setEnabled(hasChild);
    actCut->setEnabled(hasChild);

    bool selected = (activeWindow()
        && activeWindow()->textCursor().hasSelection());
    actCut->setEnabled(selected);
    actCopy->setEnabled(selected);
}
```

函数 `updateMenu()` 响应 `QWorkspace` 的信号 `QWorkspace::windowActivated()`，完成应用程序主菜单状态的更新。当工作空间部件的一个子窗口被激活的时候，将会发出 `QWorkspace::windowActivated()` 信号。

函数 `QTextCursor::hasSelection()` 判断文本编辑框的文本是否有被选中，如果有返回 `true`；否则返回 `false`。

```
CMDITextEdit* CMainWindow::createMDITextEdit(bool untitled,
    const QString& fileName)
{
    CMDITextEdit* textEdit =
        new CMDITextEdit(untitled, fileName);
    workspace->addWindow(textEdit);
    connect(textEdit, SIGNAL(updateRecentFiles()),
        this, SLOT(updateRecentFiles()));
    connect(textEdit, SIGNAL(copyAvailable(bool)),
        actCut, SLOT(setEnabled(bool)));
    connect(textEdit, SIGNAL(copyAvailable(bool)),
        actCopy, SLOT(setEnabled(bool)));
    connect(textEdit, SIGNAL(counted(int)),
        this, SLOT(showCount(int)));

    showCount(++textEdit->count);
    return textEdit;
}
```

函数 `createMDITextEdit()` 创建一个新的文本编辑框 `CMDITextEdit` 对象，并将该对象添加到工作空间部件 `QWorkspace` 中。它的第一个参数 `untitled` 指定了新文档的状态，如果 `untitled = true`，表示新的文档从来没有被保存过；而当 `untitled = false` 时，表示新的文档曾经被保存过（比如文档的“打开”操作）。第二个参数指定了文件名。

函数 `QWorkspace::addWindow()` 将新建的 `CMDITextEdit` 对象添加到工作空间部件中。



```
void CMainWindow::showCount(int count)
{
    label1->setText(tr("文档总数: %1").arg(count));
}
```

槽函数 showCount() 更新状态栏中标签窗口部件的显示内容, 设置文档总数。

现在, 创建一个新的 KDevelop 工程 workspace, 创建主程序文件 (同第 4 章的 KDevelop 工程 “designmainwindow” 的主程序完全相同), 并将第 4 章引用的查找文件类 CFindFileForm (包括 ui 文件)、Qt 设计器中绘制的主窗口的 ui 文件 (包括资源文件及其资源) 加入到工程中。最后, 将上述类 CMDITextEdit 和 CMainWindow 的定义文件和实现文件加入到工程中。

编译运行应用程序, 运行界面如图 5-19 所示。

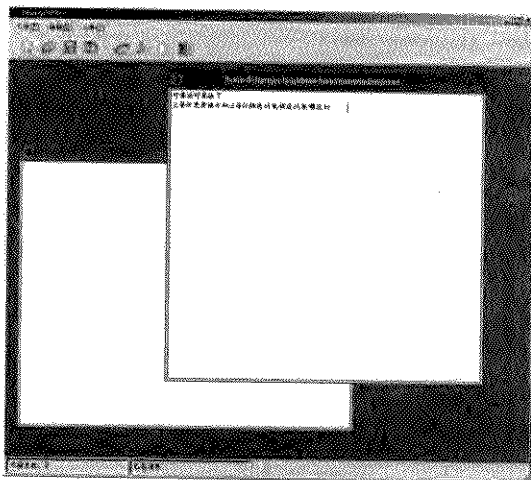


图 5-19 QWorkspace 实现 MDI 文本编辑器

5.5 多文档区部件 QMdiArea

多文档区部件 QMdiArea 提供了一个管理/显示多文档界面的区域。它通常作为应用程序多文档界面主窗口的中心部件, 实现对子窗口的管理、绘制和排布。同 QWorkSpace 不同的是, QMdiArea 具有独特的多文档子窗口类 QMdiSubWindow, 它在多文档区部件内表现为一个顶层窗口, 可以关闭、最小化和最大化, 具有独立的窗口标题。QMdiSubWindow 具有自己的布局管理器, 该布局管理器管理窗口标题栏和放置窗口部件的中心区域。多文档子窗口 QMdiSubWindow 和多文档区部件 QMdiArea 共同实现应用程序的多文档功能。通常通过调用函数 QMdiArea::addSubWindow() 为一个多文档部件添加一个多文档子窗口, 并返回该多文档子窗口的指针。

多文档区部件 QMdiArea 在 Qt 4.3 中开始引入。

下面修改上一小节中多文档功能的主窗口定义文件和实现文件中的构造函数和槽函数 doNew(), 实现用户的“新建”操作。对于其他的功能, QMdiArea 和 QWorkSpace 的实现是大同小异的, 有兴趣的读者可以试着实现一下。

主窗口类 CMainWindow 的定义文件 mainwindow.h 部分如下。

```

.....
class QMdiArea;
class CMDITextEdit;

class CMainWindow : public QMainWindow,
                    public Ui::MainWindow
{
    Q_OBJECT
public:
    CMainWindow(QWidget* = 0);

private:
    QDockWidget*   dockWidget;
    QLabel*        label1;
    QLabel*        label2;
    enum(MaxRecentFiles = 9);
    QAction*        recentFileActs[MaxRecentFiles];
    QAction*        separatorAct;
    QMdiArea*       mdiArea;
    .....
};

```

在此，将与 QWorkspace 相关的地方全部改为对 QMdiArea 的使用。

主窗口类 CMainWindow 的构造函数如下。

```

CMainWindow::CMainWindow(QWidget* parent)
: QMainWindow(parent)
{
    setupUi(this);

    mdiArea = new QMdiArea;
    setCentralWidget(mdiArea);

    connect(mdiArea, SIGNAL(subWindowActivated(QMdiSubWindow *)),
           this, SLOT(updateMenu()));

    iniDockWidget();
    iniStatusBar();
    iniConnect();
    updateRecentFiles();
    updateMenu();

    showMaximized();
    showCount(0);
}

```

创建一个 QMdiArea 对象，并设置为主窗口的中心部件。关联信号 QMdiArea::subWindowActivated() 和槽函数 CMainWindow::updateMenu()，当子窗口活动状态切换时更新菜单的状态。



```

void CMainWindow::doNew()
{
    static int sequenceNum = 1;
    CMDITextEdit* textEdit =
        createMDITextEdit(true, tr("untitled%1").arg(sequenceNum++));
    QMdiSubWindow* subWindow = mdiArea->addSubWindow(textEdit);
    subWindow->show();
}

```

当用户需要新建文档时，首先创建多文档文本编辑框 CMDITextEdit 对象 textEdit，然后为多文档区部件添加多文档子窗口，并将刚创建的 CMDITextEdit 对象 textEdit 传递给 QMdiArea::addSubWindow() 函数，该函数将会把 textEdit 对象设置为多文档子窗口的中心区域的窗口部件。

多文档区部件 QMdiArea 实现的多文档应用程序与工作空间部件 QWorkspace 的实现的多文档应用程序，界面上几乎完全相同，如图 5-20 所示。

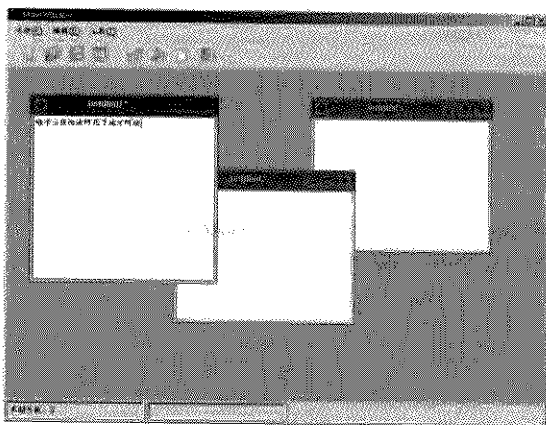


图 5-20 QMdiArea 多文档应用程序界面

注意，在关闭多文档子窗口时，不会执行 CMDITextEdit::closeEvent() 函数。这是因为当多文档子窗口被关闭时，该多文档子窗口自动被销毁了（所有子窗口部件也被销毁），并不会触发子窗口部件 CMDITextEdit 的关闭事件。如果要完成多文档子窗口关闭时保存文档的功能，一个办法是截获多文档子窗口的关闭事件（相关知识见第 11 章“事件处理”）。

5.6 小 结

Qt 布局管理是进行 GUI 图形用户界面编程的一个很重要的内容，即使很简单的界面窗口也要考虑到它的外观、窗口部件的大小和位置等。在这一章详细学习了 Qt 布局管理的内容，包括 Qt 布局管理器、分裂器以及栈部件、工作空间部件和多文档区部件等，并举例学习了它们在 GUI 编程中的应用。

截止于本章，已经结束了 Qt 初级篇的学习。初级篇共有五章，主要涉及 Qt GUI 编程的基本概念和基本知识，包括学习了 KDevelop 集成开发环境以及 Qt 设计器、rc 工具、uic 工具等。从下一章开始，将进入 Qt 中级篇的学习，学习 Qt GUI 编程的一些更高级的特性。



中 级 篇

第6章 2D绘图

第7章 拖放操作和剪贴板

第8章 文件处理

第9章 网络

第10章 多线程

第11章 事件处理

第12章 数据库

第13章 Qt的模板库和工具类

第 6 章 2D 绘图

Qt 4 中的 2D 绘图部分称为 Arthur 绘图系统，它由三个主要的类支撑起整个框架：QPainter、QPaintDevice 和 QPainterEngine。QPainter 用来执行具体的绘图相关操作，如画点、画线、填充、变换、alpha 通道等。QPaintDevice 是 QPainter 用来绘图的绘图设备，Qt 中有几种预定义的绘图设备，如 QWidget、QPixmap、QPrinter 等，它们都是从 QPaintDevice 继承。QPainterEngine 类提供了不同类型设备的接口，QPainterEngine 对程序员不透明，由 QPainter、QPaintDevice 类与其进行交互。

从 Qt 4.2 开始，Graphics View 框架取代了 Qt 3 中的 QCanvas，Graphics View 框架使用了 MVC 模式，适合对大量的 2D 图元的管理。在 Graphics View 框架中，场景 (scene) 存储了图形数据，它通过视图 (view) 以多种形式表现，每个图元 (item) 可以单独进行控制 (control)。

6.1 Arthur 绘图基础

在 Arthur 绘图框架中的基本绘图元素是画笔、画刷，本章将首先从这些最基本的绘图对象开始讲解。在讲解过程中，将贯穿一个基本的绘图程序。

6.1.1 绘图

QPainter 类具有 GUI 程序需要的绝大多数函数，能够绘制基本的图形（点、线、矩形、多边形等）以及复杂的图形（如绘图路径）。使用绘图路径 (QPaintPath) 的优点是复杂形状的图形只用生成一次，以后再绘制时只需调用 QPainter::drawPath() 就可以了。QPainterPath 对象可以用来填充、绘制轮廓。

线和轮廓使用画笔 (QPen) 进行绘制，画刷 (QBrush) 进行填充。画笔定义了风格（线形）、宽度、笔尖画刷以及端点是如何绘制的 (cap-style)、端点的连接方式 (join-style)。画刷用来填充画笔绘制的图形，可以定义不同填充模式和颜色的画刷。

当绘制文字时，字体使用 QFont 类定义。Qt 使用指定字体的属性，如果没有匹配的字体，Qt 将使用最接近的字体。字体的属性可以使用 QFontInfo 类获取。字体的度量 (measurement) 使用 QFontMetrics 类来获取。QFontDatabase 类可以得到底层窗口系统的所有可用字体的信息。

通常情况下，QPainter 以默认的坐标系统进行绘制，也可以使用 QMatrix 类对坐标进行变换。

当绘制时，可以使用 QPainter::RenderHint 来告诉绘图引擎是否启用反锯齿功能使绘图更为平滑。QPainter::RenderHint 的可取如表 6-1 中的值。

表 6-1 绘制提示

常 量	描 述
QPainter::Antialiasing	告诉绘图引擎应该在可能的情况下进行边的反锯齿绘制
QPainter::TextAntialiasing	告诉绘图引擎在可能的情况下应该绘制反锯齿的文字
QPainter::SmoothPixmapTransform	指示引擎应该使用平滑的 pixmap 变换算法（如双线性插值）而不是最近邻 (nearest neighbor) 插值算法



扩展阅读

最近邻插值算法与双线性插值算法

最近邻插值算法 (nearest neighbor) 通过对目标像素进行反向变换得到一个浮点坐标, 然后对其进行取整得到整数坐标, 这个整数坐标对应的像素值就是目的像素的像素值, 也就是浮点坐标最邻近的左上角点对应的像素值。因此, 最近邻插值算法简单, 运算量小, 但得到的图像质量不高。

双线性插值算法 (bilinear) 中, 对于一个目的像素, 设坐标通过反向变换得到的浮点坐标为 $(i+x, j+y)$, 其中 i, j 均为非负整数, x, y 为 $[0, 1)$ 区间的浮点数, 则这个像素的值 $f(i+x, j+y)$ 可由源图像中坐标为 (i, j) 、 $(i+1, j)$ 、 $(i, j+1)$ 、 $(i+1, j+1)$ 所对应的周围四个像素的值决定, 即:

$$f(i+x, j+y) = (1-x)(1-y)f(i, j) + (1-x)yf(i, j+1) + x(1-y)f(i+1, j) + xyf(i+1, j+1)$$

双线性内插值法计算量大, 但插值后图像质量较高, 不会出现像素值不连续的情况。

在 Qt 中, 绘制各种图形的函数属于 QPainter 类。QPainter 提供的绘图函数如表 6-2 所示。

表 6-2 QPainter 的绘图函数

绘图函数	绘制的图形
drawArc()	弧
drawChord()	弦
drawConvexPolygon()	凸多边形
drawEllipse()	椭圆
drawImage()	QImage 表示的图像
drawLine()	线
drawLines()	多条线
drawPath()	路径
drawPicture()	按 QPainter 指令绘制
drawPie()	扇形
drawPixmap()	QPixmap 表示的图像
drawPoint()	点
drawPoints()	多个点
drawPolygon()	多边形
drawPolyline()	多折线
drawRect()	矩形
drawRects()	多个矩形
drawRoundRect()	圆角矩形
drawText()	文字
drawTiledPixmap()	平铺图像
drawCubicBezier()	绘制三次 Bezier 曲线
drawLineSegments()	绘制折线

上面的大多数绘图函数从函数名就能够知道其功能, 可能只有 drawPicture() 函数需要解释一下。drawPicture() 函数负责绘制 QPicture 中存储的 QPainter 指令, QPicture 是可以记录 QPainter 绘图指令的类, 它将 QPainter 的绘图指令串行化为平台无关的格式存储。下面的代码将记录的绘图指令重绘:

```
QPicture picture;
picture.load("mypicture.pic");
QPainter painter(this);
painter.drawPicture(0, 0, picture);
```

上面的代码片段装入绘图文件 `mypicture.pic`，并在点 (0,0) 处重放绘图指令。也可以用 `QPicture::play()` 函数来完成相同的功能。

1. 使用画笔

Qt 使用 `QPen` 类定义画笔。画笔的属性包括线型、线宽、颜色等。画笔的属性可以在构造函数中指定，也可以使用 `setStyle()`、`setWidth()`、`setBrush()`、`setCapStyle()`、`setJoinStyle()` 等函数逐项设定画笔的各项属性。

线宽为 0 的画笔称为装饰笔。不论 `QPainter` 如何进行变换，装饰笔总是 1 个像素。

(1) 画笔风格 (Pen Style)

在 Qt 中使用 `Qt::PenStyle` 定义了 6 种画笔风格，分别是 `Qt::SolidLine`、`Qt::DashLine`、`Qt::DashDotLine`、`Qt::DashDotDotLine`、`Qt::CustomDashLine`。如图 6-1 所示。



图 6-1 画笔风格

默认的画笔风格是 `Qt::SolidLine`。实际上还有一种风格是 `Qt::NoPen`，使用 `Qt::NoPen` 时 `QPainter` 不绘制线。如果要使用自定义的线风格 (`Qt::CustomDashLine`)，则需要使用 `QPen` 的 `setDashPattern()` 函数来设置自定义风格。

下面的代码设置了一个自定义的 `QPen`。

```
QPen pen;
QVector<qreal> customDashes;
qreal blank = 4;
dashes << 2 << blank << 4 << blank << 8 << blank
    << 16 << blank << 32;
pen.setDashPattern(customDashes);
```

在上面的代码中使用了 `QVector` 定义自定义风格的画笔，在 `QVector` 中奇数位置的数是实线的长度，偶数位置的数是空白的长度。

(2) 端点风格 (Cap Style)

端点风格决定了线的端点样式，它只对线宽大于等于 1 的线有效，对装饰笔绘制的线无效。Qt 中定义了三种端点风格，用枚举类型 `Qt::PenCapStyle` 表示，分别为 `Qt::SquareCap`、`Qt::FlatCap`、`Qt::RoundCap`。`Qt::SquareCap` 类型的端点是方形的并将端点延长至线宽的一半，`Qt::FlatCap` 也是方形端点但没有将线延长，`Qt::RoundCap` 是圆形的端点。不同的端点风格如图 6-2 所示。



图 6-2 端点风格

(3) 连接风格 (Join Style)

连接风格是指两条线如何连接, 连接风格仅对线宽大于等于 1 的线有效, 对装饰笔绘制的线无效。Qt 定义了四种连接方式, 用枚举类型 `Qt::PenStyle` 表示, 分别是 `Qt::MiterJoin`, `Qt::BevelJoin`, `Qt::RoundJoin` 和 `Qt::SvgMiterJoin`。`Qt::BevelJoin` 风格削去了连接中心线前段的三角形, `Qt::MiterJoin` 填充了这个三角形, `Qt::RoundJoin` 在 `Qt::BevelJoin` 的基础上绘制了弧形使得连接平滑。`Qt::SvgMiterJoin` 是 SVG 1.2 Tiny 中定义的连接风格。默认的连接风格是 `Qt::BevelJoin`。各种连接风格如图 6-3 所示。

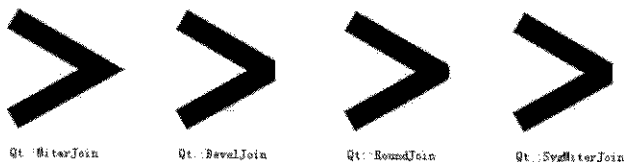


图 6-3 连接风格

下面通过一个绘图程序来讲解如何绘制简单图形, 工程名命名为 `basicdraw.pro`。示例使用调色板来设定当前画笔和接下来要讲的画刷等各项绘图元素。调色板在一个 `QDockWidget` 上生成, 其文件定义如下:

```
#ifndef PALETTE_H
#define PALETTE_H

#include <QtGui>
#include "previewlabel.h"

class Palette : public QWidget
{
    Q_OBJECT

public:
    Palette(QWidget *parent = 0);

signals:
    void penChanged(QPen& pen);
    void brushChanged(QBrush& brush);

private slots:
    void penChanged();
};
```



```
void brushChanged ();

private:
    QLabel *penColorLabel;
    QLabel *penWidthLabel;
    QLabel *penStyleLabel;
    QLabel *brushColorLabel;
    QLabel *brushStyleLabel;
    PreviewLabel *preLabel;
    QSpinBox *penWidthSpinBox;
    QComboBox *penColorComboBox;
    QComboBox *penStyleComboBox;
    QComboBox *brushColorComboBox;
    QComboBox *brushStyleComboBox;

    void createColorComboBox(QComboBox *comboBox);
    void createStyleComboBox();
};
#endif
```

在该类中定义了两个信号，当画笔发生改变时发出 `penChanged()` 信号，画刷发生改变时发出 `brushChanged()` 信号。调色板中包括用来改变线宽、画笔颜色、风格等窗口部件。

调色板类构造函数的实现如下：

```
Palette::Palette(QWidget *parent)
: QWidget(parent)
{
    penColorComboBox = new QComboBox;
    createColorComboBox(penColorComboBox);
    penColorLabel = new QLabel(tr("Pen Color:"));
    penColorLabel->setBuddy(penColorComboBox);

    penWidthSpinBox = new QSpinBox;
    penWidthSpinBox->setRange(0, 20);
    penWidthSpinBox->setSpecialValueText(tr("0 (cosmetic pen)"));

    penWidthLabel = new QLabel(tr("Pen &Width:"));
    penWidthLabel->setBuddy(penWidthSpinBox);

    createStyleComboBox();

    penStyleLabel = new QLabel(tr("&Pen Style:"));
    penStyleLabel->setBuddy(penStyleComboBox);

    brushColorComboBox = new QComboBox();
    createColorComboBox(brushColorComboBox);

    brushColorLabel = new QLabel(tr("Brush Color:"));
    brushColorLabel->setBuddy(brushColorComboBox);
```

```

brushStyleComboBox = new QComboBox;
brushStyleComboBox->addItem(tr("None"), Qt::NoBrush);

brushStyleComboBox->addItem(tr("Linear Gradient"),
    Qt::LinearGradientPattern);
brushStyleComboBox->addItem(tr("Radial Gradient"),
    Qt::RadialGradientPattern);
brushStyleComboBox->addItem(tr("Conical Gradient"),
    Qt::ConicalGradientPattern);
brushStyleComboBox->addItem(tr("Texture"), Qt::TexturePattern);

brushStyleLabel = new QLabel(tr("&Brush Style:"));
brushStyleLabel->setBuddy(brushStyleComboBox);

preLabel = new PreviewLabel(this);

connect(penColorComboBox, SIGNAL(currentIndexChanged(int)),
    this, SLOT(penChanged()));
connect(penWidthSpinBox, SIGNAL(valueChanged(int)),
    this, SLOT(penChanged()));
connect(penStyleComboBox, SIGNAL(currentIndexChanged(int)),
    this, SLOT(penChanged()));
connect(brushColorComboBox, SIGNAL(currentIndexChanged(int)),
    this, SLOT(brushChanged()));
connect(brushStyleComboBox, SIGNAL(currentIndexChanged(int)),
    this, SLOT(brushChanged()));
connect(this, SIGNAL(penChanged(QPen&)),
    preLabel, SLOT(penChanged(QPen&)));
connect(this, SIGNAL(brushChanged(QBrush&)),
    preLabel, SLOT(brushChanged(QBrush&)));

QGridLayout *mainLayout = new QGridLayout;
mainLayout->addWidget(penColorLabel, 0, 0, Qt::AlignRight);
mainLayout->addWidget(penColorComboBox, 0, 1);
mainLayout->addWidget(penWidthLabel, 1, 0, Qt::AlignRight);
mainLayout->addWidget(penWidthSpinBox, 1, 1);
mainLayout->addWidget(penStyleLabel, 2, 0, Qt::AlignRight);
mainLayout->addWidget(penStyleComboBox, 2, 1);
mainLayout->addWidget(brushColorLabel, 3, 0, Qt::AlignRight);
mainLayout->addWidget(brushColorComboBox, 3, 1);
mainLayout->addWidget(brushStyleLabel, 4, 0, Qt::AlignRight);
mainLayout->addWidget(brushStyleComboBox, 4, 1);
mainLayout->addWidget(preLabel, 5, 0, 6, 2);
setLayout(mainLayout);

penChanged();
brushChanged();

setWindowTitle(tr("Basic Drawing"));
}

```



在构造函数中初始化了每个选择项的条目。注意设置线宽的初始值用了

```
penWidthSpinBox->setSpecialValueText(tr("0 (cosmetic pen)"));
```

setSpecialValue()函数是 QSpinBox 类的特殊用法, 用来在 QSpinBox 中显示文字而不是默认的数值。

画笔线型组合框中的每个条目是使用不同风格绘制的直线, 这需要一个单独的函数 createStyleComboBox()来创建, 后面将会讲到。

接下来是画笔各属性改变时发出信号通知相应对象的函数:

```
void Palette::penChanged()
{
    QPen pen;

    int width = penWidthSpinBox->value();
    pen.setWidth(width);

    QColor color = penColorComboBox->itemData(
        penColorComboBox->currentIndex(), Qt::UserRole).value<QColor>();
    pen.setColor(color);

    Qt::PenStyle penStyle = (Qt::PenStyle)penStyleComboBox->itemData(
        penStyleComboBox->currentIndex(), Qt::UserRole).toInt();
    pen.setStyle(penStyle);

    emit penChanged(pen);
}
```

由于要在改变画笔颜色组合框的条目上加上相应的颜色的标识, 所以在创建画笔颜色组合框时要进行一些特殊处理。

```
void Palette::createColorComboBox(QComboBox *comboBox)
{
    QPixmap pix(16, 16);

    QPainter pt(&pix);
    pt.fillRect(0, 0, 16, 16, Qt::black);
    comboBox->addItem(QIcon(pix), tr("black"), Qt::black);
    pt.fillRect(0, 0, 16, 16, Qt::red);
    comboBox->addItem(QIcon(pix), tr("red"), Qt::red);
    pt.fillRect(0, 0, 16, 16, Qt::green);
    comboBox->addItem(QIcon(pix), tr("green"), Qt::green);
    pt.fillRect(0, 0, 16, 16, Qt::blue);
    comboBox->addItem(QIcon(pix), tr("blue"), Qt::blue);
    pt.fillRect(0, 0, 16, 16, Qt::yellow);
    comboBox->addItem(QIcon(pix), tr("yellow"), Qt::yellow);
    pt.fillRect(0, 0, 16, 16, Qt::cyan);
    comboBox->addItem(QIcon(pix), tr("cyan"), Qt::cyan);
    pt.fillRect(0, 0, 16, 16, Qt::magenta);
    comboBox->addItem(QIcon(pix), tr("magenta"), Qt::magenta);
}
```

在上面的函数中, 分别为 7 种颜色生成了 1 幅 16×16 的 QPixmap 图像, 图像就是相应颜色填充的矩形。该图像直接显示在组合框的条目中。

由于要在组合框的每个条目上显示画笔线型的实际效果, 所以要实现组合框条目的代理类, 让代理类进行实际的条目绘制。创建线型组合框的函数实现如下:

```
void Palette::createStyleComboBox()
{
    penStyleComboBox = new QComboBox;
    penStyleComboBox->setItemDelegate(
        new QPenStyleDelegate((QObject *)penStyleComboBox));
    penStyleComboBox->addItem(tr("Solid"), Qt::SolidLine);
    penStyleComboBox->addItem(tr("Dash"), Qt::DashLine);
    penStyleComboBox->addItem(tr("Dot"), Qt::DotLine);
    penStyleComboBox->addItem(tr("Dash Dot"), Qt::DashDotLine);
    penStyleComboBox->addItem(tr("Dash Dot Dot"), Qt::DashDotDotLine);
    penStyleComboBox->addItem(tr("None"), Qt::NoPen);
}
```

组合框条目的线型代理类 QPenStyleDelegate 用来完成条目的绘制, 以实现特殊效果。线型代理类定义如下所示:

```
#ifndef QPENSTYLEDELEGATE_H
#define QPENSTYLEDELEGATE_H

#include <QtGui>

class QPenStyleDelegate : public QAbstractItemDelegate
{
    Q_OBJECT

public:
    QPenStyleDelegate(QObject *parent = 0);

    void paint(QPainter *painter,
               const QStyleOptionViewItem &option,
               const QModelIndex &index) const;

    QSize sizeHint(const QStyleOptionViewItem &option,
                   const QModelIndex &index) const;
};
#endif
```

代理类中的 paint() 函数负责条目的绘制, sizeHint() 确定条目的大小。代理类的绘图函数实现如下:

```
void QPenStyleDelegate::paint(QPainter *painter,
                              const QStyleOptionViewItem &option,
                              const QModelIndex &index) const
{
    QString text = index.data(Qt::DisplayRole).toString();
    Qt::PenStyle penStyle = (Qt::PenStyle)index.data(Qt::UserRole).toInt();
    QRect r = option.rect;
```



```
if (option.state & QStyle::State_Selected) {  
    painter->save();  
    painter->setBrush(option.palette.highlight());  
    painter->setPen(Qt::NoPen);  
    painter->drawRect(option.rect);  
    painter->setPen(QPen(option.palette.highlightedText(), 2,  
        penStyle));  
}  
else  
    painter->setPen(penStyle);  
painter->drawLine(0, r.y() + r.height()/2,  
    r.right(), r.y() + r.height()/2);  
  
if (option.state & QStyle::State_Selected)  
    painter->restore();  
}
```

绘制函数中要对选项是否被选中进行判断, 如果选中, 则需要高亮显示。对于条目的大小, 简单地返回宽 100 高 30 的矩形作为一个条目所占的空间, 代码如下:

```
QSize QPenStyleDelegate::sizeHint(const QStyleOptionViewItem &option,  
    const QModelIndex &index) const  
{  
    return QSize(100, 30);  
}
```

这里生成了画笔的调色板, 具体绘制在后面将会逐步展开。

2. 画刷

在 Qt 中图形使用 `QBrush` 填充。画刷包括填充颜色和风格(填充模式)。在 Qt 中, 颜色使用 `QColor` 类表示。`QColor` 支持 RGB、HSV、CMYK 颜色模型。`QColor` 还支持 alpha 混合的轮廓和填充(可以实现透明效果), `QColor` 类与平台、设备无关(通过 `QColorMap` 类和硬件进行映射)。

填充模式由 `Qt::BrushStyle` 枚举变量定义, 包括基本模式填充、渐变填充和纹理填充。基本模式填充包括由各种点、线组合的模式。下面的例子将展示 Qt 中预定义的填充模式, 在这个例子中, 通过窗口部件上用各种模式填充不同的矩形来观察填充效果。



扩展阅读

QColor 使用的颜色模型

Qt 支持 RGB、HSV 和 CMYK 颜色模型。RGB 是面向硬件的模型, 颜色由红、绿、蓝三种基色混合而成。HSV 模型则比较符合人对颜色的感觉, 由色调(0-359)、饱和度(0-255)、亮度(0-255)组成。CMYK 由青、洋红、黄、黑四种基色组成, 主要用在打印机等硬拷贝设备上, 每个颜色分量的取值为 0~255。另外 `QColor` 还可以使用 SVG 1.0 中定义的任何颜色名为参数初始化。

该工程名为 `paintertest.pro`, 演示了画笔、画刷等种绘图元素, 其中画刷填充模式的绘制函数如下:


```

void TestWidget::paintBrushStyle(QPainter& painter)
{
    list.clear();
    list << "Qt::NoBrush" << "Qt::SolidPattern" << "Qt::Dense1Pattern"
        << "Qt::Dense2Pattern" << "Qt::Dense3Pattern"
        << "Qt::Dense4Pattern" << "Qt::Dense5Pattern"
        << "Qt::Dense6Pattern" << "Qt::Dense7Pattern"
        << "Qt::HorPattern" << "Qt::VerPattern" << "Qt::CrossPattern"
        << "Qt::BDiagPattern" << "Qt::FDiagPattern"
        << "Qt::DiagCrossPattern";
    QBrush brush;
    QRect rect;
    int x = 10;
    int y = 10;
    for(int i=Qt::NoBrush; i <= Qt::DiagCrossPattern; i++)
    {
        brush.setStyle((Qt::BrushStyle)i);
        painter.setBrush(brush);
        rect.setRect(x, y, w, h);
        painter.drawRect(rect);
        painter.drawText(x, y + h * 15, list.at(i));
        if((i+1)%4 == 0) {
            x = 10;
            y += h + h/2;
        }
        else
            x += w + w/2;
    }
}

```

函数的 paintEvent 通过变换矩形的位置和填充模式，绘出了各种填充模式。运行效果如图 6-4 所示。

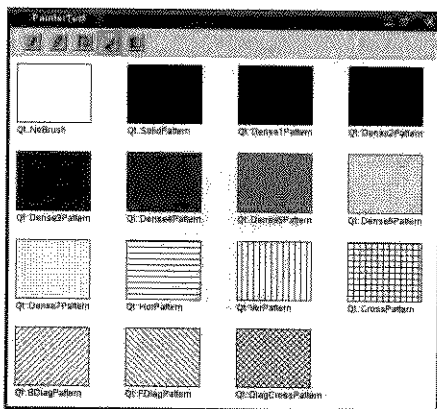


图 6-4 Qt 基本填充模式

Qt 4 提供了渐变填充的画刷。渐变填充包括两个要素：颜色的变化和路径的变化。颜色变化可以指定从一种颜色渐变到另外一种颜色，也可以在变化的路径上指定一些点的颜色进行分段渐变。在 Qt 4 中，提供了三种渐变填充：线性（`QLinearGradient`）、圆形（`QRadialGradient`）和圆锥渐变（`QConicalGradient`）。所有的类均从 `QGradient` 类继承。

线性渐变填充指定两个控制点，画刷在两个控制点之间进行颜色插值。通过创建 `QLinearGradient` 对象来设置画刷。如下例所示：

```
QLinearGradient linearGradient(0, 0, 200, 100);
linearGradient.setColorAt(0, Qt::red);
linearGradient.setColorAt(0.5, Qt::green);
linearGradient.setColorAt(1, Qt::blue);
painter.setBrush(linearGradient);
painter.drawRect(0, 0, 200, 100);
```

这段代码生成的填充如图 6-5 所示。

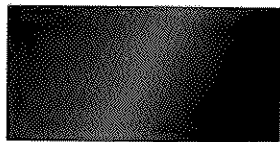


图 6-5 线性渐变填充

在 `QGradient` 构造函数中指定线性填充的两点分别为(0,0)、(100,100)。 `setColorAt()` 函数在 0~1 的范围内设置指定位置的颜色。

圆形渐变填充需要指定圆心，半径和焦点，画刷在焦点和圆上的所有点之间进行颜色插值。创建 `QRadialGradient` 对象来设置画刷，如下例所示。

```
QRadialGradient radialGradient(50, 50, 50, 30, 30);
radialGradient.setColorAt(0.2, Qt::cyan);
radialGradient.setColorAt(0.8, Qt::yellow);
radialGradient.setColorAt(1, Qt::magenta);
painter.setBrush(radialGradient);
painter.drawEllipse(0, 0, 100, 100);
```

上面的代码填充效果如图 6-6 所示。

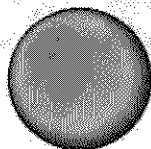


图 6-6 圆形渐变填充

圆锥渐变填充指定圆心和开始角，画刷沿圆心逆时针对颜色进行插值。创建 `QConicalGradient` 对象并设置画刷。下面的代码展示了圆锥渐变填充：

```
QConicalGradient conicalGradient(60, 40, 30);
```

```
conicalGradient.setColorAt(0, Qt::gray);
conicalGradient.setColorAt(0.4, Qt::darkGreen);
conicalGradient.setColorAt(0.6, Qt::darkMagenta);
conicalGradient.setColorAt(1, Qt::darkBlue);
painter.setBrush(conicalGradient);
painter.drawEllipse(0, 0, 100, 100);
```

这段代码填充效果如图 6-7 所示。



图 6-7 圆锥渐变填充

为了实现自定义填充，还可以使用使用 QPixmap 或 QImage 对象进行纹理填充。两种图像分别使用 setTexture() 和 setTextureImage() 函数加载纹理。

下面通过将上一节的基本图形绘制程序完善来说明绘图过程。现在实现当调色板中用户改变当前画刷的响应槽函数如下：

```
void Palette::brushChanged()
{
    QBrush brush;

    QColor color = brushColorComboBox->itemData(
        brushColorComboBox->currentIndex(),
        Qt::UserRole).value<QColor>();

    Qt::BrushStyle style = Qt::BrushStyle(brushStyleComboBox->itemData(
        brushStyleComboBox->currentIndex(), Qt::UserRole).toInt());

    if (style == Qt::LinearGradientPattern) {
        QLinearGradient linearGradient(0, 0, 100, 100);
        linearGradient.setColorAt(0.0, Qt::white);
        linearGradient.setColorAt(0.2, Qt::green);
        linearGradient.setColorAt(1.0, Qt::black);
        brush = linearGradient;
    } else if (style == Qt::RadialGradientPattern) {
        QRadialGradient radialGradient(50, 50, 50, 70, 70);
        radialGradient.setColorAt(0.0, Qt::white);
        radialGradient.setColorAt(0.2, Qt::green);
        radialGradient.setColorAt(1.0, Qt::black);
        brush = radialGradient;
    } else if (style == Qt::ConicalGradientPattern) {
        QConicalGradient conicalGradient(50, 50, 150);
        conicalGradient.setColorAt(0.0, Qt::white);
        conicalGradient.setColorAt(0.2, Qt::green);
```



```

        conicalGradient.setColorAt(1.0, Qt::black);
        brush = conicalGradient;
    } else if (style == Qt::TexturePattern) {
        brush = QBrush(QPixmap(":/images/ellipse.png"));
    } else {
        brush.setColor(color);
        brush.setStyle(style);
    }
    emit brushChanged(brush);
}

```

函数根据当前的画刷模式和画刷颜色生成了相应的画刷，对于渐变画刷和纹理画刷，则生成了一个固定的渐变模式和纹理。读者可自行将其修改为用户可改变的渐变模式和纹理。

为了能够实时地看到选择调色板的效果，使用了一个 QLabel 对象作为预览区域，定义如下：

```

#ifndef PREVIEWLABEL_H
#define previewlabel

#include <QtGui>

class PreviewLabel : public QLabel
{
    Q_OBJECT

public:
    PreviewLabel(QWidget *parent = 0);

public slots:
    void penChanged(QPen& pen);
    void brushChanged(QBrush& brush);

protected:
    void paintEvent(QPaintEvent *event);

private:
    QPen curPen;
    QBrush curBrush;
};

#endif

```

当画笔和画刷改变时，预览标签将显示实际的效果。预览类的实现文件如下：

```

#include "previewlabel.h"

PreviewLabel::PreviewLabel(QWidget *parent)
    : QLabel(parent)
{
}

```

```

void PreviewLabel::penChanged(QPen& pen)
{
    curPen = pen;
    update();
}

void PreviewLabel::brushChanged(QBrush& brush)
{
    curBrush = brush;
    update();
}

void PreviewLabel::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.setPen(curPen);
    painter.setBrush(curBrush);
    painter.drawRect(rect().x() + 10, rect().y() + 10, rect().width() - 20,
        rect().height() - 20);
}

```

在预览类中，接收画笔和画刷改变的信号，并更新预览区域。程序的运行效果如图 6-8 所示。

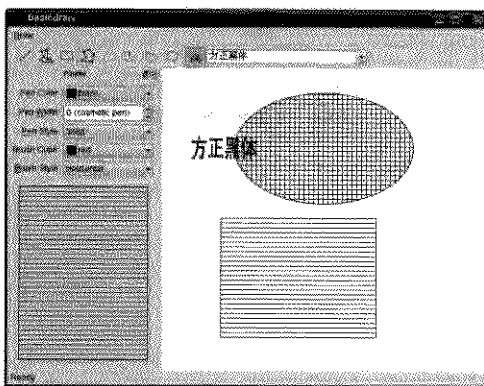


图 6-8 2D 绘图

3. 双缓冲绘图

在 Qt 4 中，所有的窗口部件默认都使用双缓冲进行绘图。使用双缓冲，可以减轻绘制的闪烁感。在有些情况下，用户需要关闭双缓冲，自己管理绘图。下面的语句设置了窗口部件的 Qt::WA_PaintOnScreen 属性，就关闭了窗口部件的双缓冲。

```
myWidget->setAttribute(Qt::WA_PaintOnScreen);
```

由于 Qt 4 不再提供异或笔，组合模式 QPainter::CompositionMode_Xor（6.6 节介绍）并不是异或笔，Qt 4 只提供了 QRubberBand 实现矩形和直线的绘图反馈。因此要实现在绘制过程中动态反馈必须采用其



他方法。程序中使用双缓冲来解决这个问题。在绘图过程中，一个缓冲区绘制临时内容，一个缓冲区保存绘制好的内容，最后进行合并。

在交互绘制过程中，程序将图像缓冲区复制到临时缓冲区，并在临时缓冲区上绘制，绘制完毕再将结果复制到图像缓冲区。如果没有交互绘制，则直接将图像缓冲区显示到屏幕上。

双缓冲绘图的流程如图 6-9 所示。

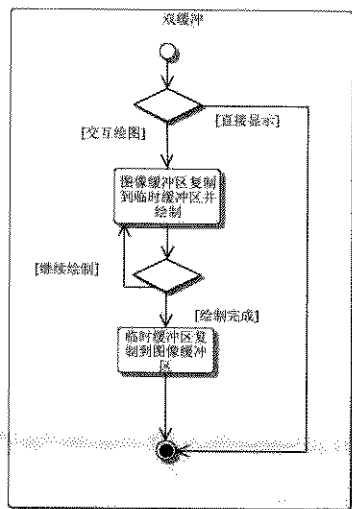


图 6-9 双缓冲绘图的流程

用于绘图的窗口部件定义如下：

```
#ifndef FORM_H
#define FORM_H

#include <QtGui>

class Form : public QWidget
{
    Q_OBJECT

public:
    Form(QWidget * parent = 0);
    enum ShapeType {
        Line,
        Polyline,
        Rectangle,
        Polygon,
        Arc,
        Pie,
```

```

        Chord,
        Ellipse,
        Text
    };

    void setShape(ShapeType shape);

public slots:
    void fontChanged(const QFont& font);
    void penChanged(QPen& pen);
    void brushChanged(QBrush& brush);

protected:
    bool bDrawing;
    int x,y,w,h;
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);

private:
    QImage bufferImage;
    QImage tempImage;
    ShapeType curShape;
    QPen curPen;
    QBrush curBrush;
    QFont textFont;
    int thickness;

    void paint(QImage& image);
};
#endif

```

在头文件中定义了枚举类型 `ShapeType`，用来记录当前绘制的图形。用来绘图的窗口部件构造函数如下：

```

Form::Form(QWidget * parent)
: QWidget(parent)
{
    setAttribute(Qt::WA_NoBackground);
    bDrawing = false;
    curShape = Ellipse;

    resize(800,600);
    bufferImage = QImage(width(), height(),
        QImage::Format_ARGB32_Premultiplied);
    bufferImage.fill(qRgb(255, 255, 255));
    tempImage = QImage(width(), height(),
        QImage::Format_ARGB32_Premultiplied);
}

```



在构造函数中初始化了两个缓冲区: `bufferImage` 用来存储最终的图形, `tempImage` 是临时缓冲区。`setShape()` 函数设置当前绘制的图形种类。

```
void Form::setShape(ShapeType shape)
{
    curShape = shape;
}
```

在 `mousePressEvent()` 函数中, 当鼠标按下时, 设置 `bDrawing` 值为 “true”, 表示进入交互绘图状态。

```
void Form::mousePressEvent( QMouseEvent * event )
{
    bDrawing = true;
    x = event->x();
    y = event->y();
}
```

`mouseMoveEvent()` 函数处理鼠标移动时将图形缓冲区中的内容复制到临时缓冲区, 并绘制临时的反馈图形。

```
void Form::mouseMoveEvent( QMouseEvent *event)
{
    w = event->x() - x;
    h = event->y() - y;
    tempImage = bufferImage;
    paint(tempImage);
}
```

`mouseReleaseEvent()` 函数处理鼠标按钮释放时, 在绘图缓冲区上绘制图形。

```
void Form::mouseReleaseEvent( QMouseEvent *event)
{
    bDrawing = false;
    paint(bufferImage);
}
```

`paintEvent()` 函数根据当前是否在绘图, 在不同的缓冲区上绘制图形。

```
void Form::paintEvent( QPaintEvent *event)
{
    QPainter painter(this);
    if(bDrawing)
        painter.drawImage(QPoint(0, 0), tempImage);
    else
        painter.drawImage(QPoint(0, 0), bufferImage);
}
```

实际的绘制函数 `paint()` 只实现了矩形、椭圆和文字的绘制, 读者可自行完成其他图形的绘制。

```
void Form::paint( QImage& image)
{
    QPainter painter(&image);
```



```

painter.setPen(curPen);
painter.setBrush(curBrush);
switch(curShape) {
case Rectangle:
    painter.drawRect(x, y, w, h);
    break;
case Ellipse:
    painter.drawEllipse(x, y, w, h);
    break;
case Text:
    QFontMetrics metrics(textFont);
    QRect rect = metrics.boundingRect(textFont.family());
    painter.setFont(textFont);
    painter.translate(x, y);
    painter.scale(w/rect.width(), h/rect.height());
    painter.drawText(0, rect.height(), textFont.family());
    break;
}
update();
}

```

其他的一些绘图要素变化的槽函数如下:

```

void Form::fontChanged(const QFont& font)
{
    textFont = font;
}

void Form::penChanged(QPen& pen)
{
    curPen = pen;
}

void Form::brushChanged(QBrush& brush)
{
    curBrush = brush;
}

```

主窗口负责建立菜单、工具条及绘图区, 主窗口的定义如下:

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QActionGroup>
#include <QFontComboBox>
#include "form.h"
#include "palette.h"

class MainWindow : public QMainWindow

```



```
{
    Q_OBJECT

public:
    MainWindow();

private slots:
    void draw(QAction* action);

private:
    void createActions();
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createDockWindows();

    Palette *paletteWidget;
    Form *form;

    QMenu *drawMenu;
    QToolBar *drawToolBar;
    QFontComboBox *fontCmb;

    QAction *rectangleAct;
    QAction *ellipseAct;
    QAction *textAct;
    .....
    QActionGroup *drawActGroup;
};
#endif
```

主窗口的实现程序如下:

```
#include <QtGui>
#include "mainwindow.h"
// 构造函数创建菜单、工具条等元素
MainWindow::MainWindow()
{
    resize(800,600);
    form = new Form(this);
    setCentralWidget(form);

    createActions();
    createMenus();
    createToolBars();
    createStatusBar();
    createDockWindows();
}
// 设置当前绘制图形的种类
```

```

void MainWindow::draw(QAction* action)
{
    if(action == rectangleAct)
        form->setShape(Form::Rectangle);
    form->setShape(Form::Ellipse);
    else if(action == textAct)
        form->setShape(Form::Text);
    ....
}

// 创建不同图形的 QAction，并将所有绘图 QAction 归到一个 QActionGroup 中
void MainWindow::createActions()
{
    rectangleAct = new QAction(QIcon(":/images/rectangle.png"),
        tr("&rectangle"), this);
    rectangleAct->setCheckable(true);

    ellipseAct = new QAction(QIcon(":/images/ellipse.png"), tr("&Ellipse"),
        this);
    ellipseAct->setCheckable(true);
    ellipseAct->setChecked(true);

    textAct = new QAction(QIcon(":/images/text.png"), tr("&Text"), this);
    textAct->setCheckable(true);
    ....

    drawActGroup = new QActionGroup(this);
    drawActGroup->addAction(rectangleAct);
    drawActGroup->addAction(ellipseAct);
    drawActGroup->addAction(textAct);
    ....
    connect(drawActGroup, SIGNAL(triggered(QAction*)), this,
        SLOT(draw(QAction*)));
}

// 创建菜单
void MainWindow::createMenus()
{
    drawMenu = menuBar()->addMenu(tr("&Draw"));
    drawMenu->addAction(rectangleAct);
    drawMenu->addAction(ellipseAct);
    drawMenu->addAction(textAct);
    ....
}

// 创建工具条
void MainWindow::createToolBars()
{
    drawToolBar = addToolBar(tr("Draw"));
    drawToolBar->addAction(rectangleAct);
    drawToolBar->addAction(ellipseAct);

```



```
drawToolBar->addSeparator();
drawToolBar->addAction(textAct);
fontCmb = new QFontComboBox(drawToolBar);
drawToolBar->addWidget(fontCmb);
connect(fontCmb, SIGNAL(currentFontChanged(const QFont&)), form,
        SLOT(fontChanged(const QFont&)));
fontCmb->setCurrentFont(font());
}
// 创建状态条
void MainWindow::createStatusBar()
{
    statusBar()->showMessage(tr("Ready"));
}
// 创建调色板
void MainWindow::createDockWindows()
{
    QDockWidget *dock = new QDockWidget(tr("Palette"), this);
    dock->setAllowedAreas(Qt::LeftDockWidgetArea |
        Qt::RightDockWidgetArea);
    paletteWidget = new Palette(dock);
    dock->setWidget(paletteWidget);
    addDockWidget(Qt::LeftDockWidgetArea, dock);

    connect(paletteWidget, SIGNAL(penChanged(QPen&)), form,
        SLOT(penChanged(QPen&)));
    connect(paletteWidget, SIGNAL(brushChanged(QBrush&)), form,
        SLOT(brushChanged(QBrush&)));
}
```

4. 使用 Alpha 通道

在 Windows、Mac OS X 和有 X Render 扩展的 X11 系统上，Qt 4 能够支持 Alpha 通道。通过使用 Alpha 通道，可以实现半透明效果。QColor 类中定义了 Alpha 通道的透明度，0 表示完全透明，255 表示完全不透明，默认情况下是完全不透明。

下面实现一个小程序，该程序显示一个随鼠标移动的半透明提示框。程序代码如下：

```
#include <QApplication>
#include <QtGui>
#include <QTextCodec>

class MyWidget : public QWidget
{
public:
    MyWidget(QWidget *parent = 0);
protected:
    void mouseMoveEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);
private:
```

```

    int x,y;
    QPixmap pixmap;
    QPixmap background;
};
// 在构造函数中装入背景图
MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    resize(800,600);
    pixmap = QPixmap(100,50);
    background = QPixmap("background.bmp");
    x = -1;
    y = -1;
}
// 鼠标移动时形成一个 pixmap
void MyWidget::mouseMoveEvent(QMouseEvent *event)
{
    x = event->x();
    y = event->y();
    pixmap.fill(QColor(255,255,255,127));
    QPainter painter(&pixmap);
    painter.setPen(QColor(255,0,0));
    painter.drawText(20, 40, QString("%1").arg(x) + ", " +
        QString("%1").arg(y));
    update();
}
// 绘制背景图和透明的 pixmap
void MyWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.drawPixmap(0, 0, background);
    painter.drawPixmap(x, y, pixmap);
}

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
    MyWidget widget;
    widget.setMouseTracking(true);
    widget.show();
    return app.exec();
}

```

在程序中，定义了一个背景是半透明的 QPixmap 对象 pixmap，随着鼠标的移动显示鼠标指针的 (x, y) 坐标值。主程序中的“setMouseTracking(true)”是打开鼠标移动跟踪，默认情况下只有在鼠标按下后才发送 QMouseEvent() 事件，打开鼠标移动跟踪后就能够随时发送了。图 6-10 窗口中的半透明矩形显示了鼠标位置。



图 6-10 Alpha 效果提示框

注意 QWidget 类有一个属性 `windowOpacity`，通过 `setWindowOpacity(qreal level)` 可以设置窗口的透明度。但该属性和这里讨论的 Alpha 通道实现原理并不相同，Qt4 在 Windows 和 Mac OS X 平台上直接支持该属性，但在 X11 平台上却需要 Composite 扩展才能工作（Alpha 通道使用的是 X11 的 XRender 扩展）。

6.1.2 绘图设备

QPaintDevice 类是实际绘制设备的基类。QPainter 能够在 QPaintDevice 子类上进行绘制，如 QWidget、QImage、QPixmap、QGLWidget、QGLPixelBuffer、QPicture、QPrinter 和 QSvgGenerator。

要实现自己的绘图设备，必须从 QPaintDevice 类继承并实现虚函数 `QPaintDevice::paintEngine()` 以告知 QPainter 能够在这个特定的设备上绘制图形。同时还需要从 QPaintEngine 类继承自定义的图形绘制引擎。

几种常用绘图设备说明如下：

1. Widget

QWidget 是所有用户界面元素的基类。窗口部件是用户界面的原子元素，它接收鼠标、键盘和窗口系统的其他事件并在屏幕上绘制自己。

2. Image

QImage 类提供了与硬件无关的图像表示，它为直接操作像素提供优化。QImage 支持单色、8-bit、32-bit 和 alpha 混合图像。使用 QImage 的优点是可以获得与平台无关的绘制操作，另外还有一个好处是图像可以不必在 GUI 线程中处理。

3. QPixmap

QPixmap 是后台显示的图像，它为在屏幕上显示图像提供优化。不同于 QImage，pixmap 的图像数据是用户不可见，而且由底层窗口系统管理。为了优化 QPixmap 绘图，Qt 提供了 QPixmapCache 类来存储临时的 pixmap。

Qt 还提供了从 QPixmap 继承的 QBitmap 类。QBitmap 表示单色的 pixmap，主要用来创建自定义的 QCursor 和 QBrush 对象、构造 QRegion 对象、设置 pixmap 和窗口部件的掩码。

4. OpenGL Widget

Qt 提供了 QtOpenGL 模块来实现 OpenGL 操作。QGLWidget 允许使用 OpenGL API 进行绘制。同时 QGLWidget 是 QWidget 的子类，因此 QPainter 也可以在上面绘制。这样可以使 Qt 能够利用 OpenGL 完成绘制操作，如变换和绘制 pixmap。

5. Pixel Buffer

QGLPixelBuffer 从 QPaintDevice 继承，封装了 OpenGL pbuffer。使用 pbuffer 绘制通常是全硬件加速，这比使用 QPixmap 绘制更迅速。

6. FrameBuffer

QGLFrameBufferObject 从 QPaintDevice 继承，QGLFrameBufferObject 封装了 OpenGL framebuffer 对象。FrameBuffer 对象用来实现后台屏幕绘制，比 pixel buffer 更好一些。

7. Picture

QPicture 类是能够记录和重演 QPainter 命令的绘图设备。Picture 串行化 painter 的命令为平台无关的格式。QPicture 同时也与分辨率无关，如 QPicture 能够在不同的设备上（如 svg、pdf、ps、打印机和屏幕）有一致的显示。QPicture::load() 和 QPicture::save() 函数分别完成载入和存储图像。

8. Printer

QPrinter 类是在打印机上绘制的绘图设备。在 Windows 和 Mac OS X 上，QPrinter 使用内建的打印机驱动程序。在 X11 上，QPrinter 产生 postscript 代码并发送给 lpr、lp 或其他打印程序。QPrinter 可以在任意其他的 QPrintEngine 对象上打印，也可以直接生成 PDF 文件。

QPrintEngine 类定义了 QPrinter 如何和其他打印机系统交互的接口。当要创建自己的打印引擎时，可以从 QPaintEngine 和 QPrintEngine 上继承。

6.2 坐标系统与坐标变换

6.2.1 坐标系统

Qt 的坐标由 QPainter 控制，同时也由 QPaintDevice 和 QPaintEngine 类控制。QPaintDevice 类是绘图设备的基类，QWidget、QPixmap、QImage 和 QPrinter 都是 QPaintDevice 的子类。Qt 的绘图设备默认坐标系统是坐标原点在左上角，X 轴向右增长，Y 轴向下增长。默认的单位在基于像素的设备上是像素，在打印机上则是 1/72 英寸（约等于 0.35 毫米）。

QPainter 的逻辑坐标与 QPaintDevice 的物理坐标之间的映射由 QPainter 的变换矩阵、视口和窗口处理。逻辑坐标与物理坐标默认是一致的。QPainter 也支持坐标变换（如旋转和伸缩）。

6.2.2 坐标变换

通常 QPainter 在设备的坐标系统上绘制图形，但 QPainter 也支持坐标变换。可以通过 QPainter::scale() 函数进行比例变换；可以使用 QPainter::rotate() 函数进行旋转变换；平移变换则使用



`QPainter::translate()`函数; `QPainter::shear()`函数对图形进行扭曲操作。所有变换操作的变换矩阵都可以通过 `QPainter::worldMatrix()`函数取出。不同的变换矩阵可以使用堆栈保存,用 `QPainter::save()`保存变换矩阵到堆栈,用 `QPainter::restore()`函数将其弹出堆栈。

为了实现更复杂的变化,可以使用 `QMatrix` 定义坐标系统的二维变换。`QMatrix` 对象实际上是一个 3×3 的变换矩阵,其模型如图 6-11 所示。

$m11$	$m12$	0
$m21$	$m22$	0
dx	dy	1

图 6-11 `QMatrix` 变换矩阵

其中 dx 、 dy 表示水平和垂直偏移量, $m11$ 和 $m22$ 表示水平和垂直方向的比例。 $m12$ 和 $m21$ 表示水平和垂直方向的扭曲程度。

`QMatrix` 将一点变换为另外一点的公式如下所示:

$$x' = m11 \times x + m21 \times y + dx$$

$$y' = m22 \times y + m12 \times x + dy$$

其中 (x,y) 是变换前的点, (x',y') 是变换后的点。

矩阵可以通过 `setMatrix()`函数进行设置,然后可以使用 `translate()`、`rotate()`、`scale()`和 `shear()`等函数进行变换。

Qt 4.3 中引入了 `QTransform` 类来表示变换矩阵。与 `QMatrix` 不同的是, `QTransform` 支持透视变换。使用 `toAffine()`函数可以将 `QTransform` 对象转换为 `QMatrix` 对象,但这将丢失 `QTransform` 的透视变换数据。

在 `QPainter` 上绘制使用的是逻辑坐标,Qt 再将其转换为绘图设备上的物理坐标。逻辑坐标与物理坐标的变换由 `QPainter` 的 `worldMatrix()`函数,以及 `QPainter` 的 `viewport()`和 `window()`函数处理。视口表示物理坐标下的任意矩形,而窗口表示在逻辑坐标下的相同矩形。默认情况下逻辑坐标与物理坐标是相同的,与绘图设备上的矩形也是一致的。

使用窗口—视口变换可以使逻辑坐标符合自定义要求。这个机制通常用来完成与设备无关的绘图代码。例如,可以设置逻辑坐标 $(-100,-100)$ 到 $(100,100)$ 且原点在 $(0,0)$,通过调用 `QPainter::setWindow()` 函数可以完成这个操作:

```
QPainter painter(this);  
painter.setWindow(QRect(-100, -100, 200, 200));
```

现在,逻辑坐标的 $(-100,-100)$ 对应着绘图设备的物理坐标 $(0,0)$ 。这样可以独立于绘图设备,始终在指定的逻辑坐标上工作。

设置窗口或视口矩形实际上是执行线性变换。本质上是窗口四个角映射到相应的视口四个角,反之亦然。因此保持视口和窗口 X 轴和 Y 轴之间的变换比例一致,保证变换没有变形。

窗口—视口变换只是线性变换,不执行裁剪等操作。例如当绘制超出了窗口时,这些绘制仍会通过线性变换映射到视口进行绘制。

Qt 的绘制过程是先进进行坐标变换,再进行窗口—视口变换。

6.3 用不同的字体

Qt 提供了 QFont 类来表示字体。当创建 QFont 对象时, Qt 会使用指定的字体, 如果没有对应的字体, Qt 将寻找一种最接近的已安装字体。字体信息可以通过 QFontInfo 取出, 并可用 QFontMetrics 取得字体的相关数据。函数 exactMatch() 判断底层窗口系统中是否有完全对应的字体。

使用 QApplication::setFont() 可以设置应用程序的默认字体。如果选择的字体不包括所有要显示的字符, QFont 将会尝试寻找最接近的字体。当 QPainter 绘制指定字体中不存在的字符时, 将绘制一个空心的正方形。

下面编写一个小程序将系统的所有字体显示出来。程序的主窗口定义如下:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QtGui>

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);

protected slots:
    void changeFontColor();
    void fontSizeChanged(const QString &text);

private:
    void createAction();
    void createMenus();
    void createToolBars();
    void listAllFonts();
    void updateColor();

    QScrollArea *scrollArea;
    QTextEdit *fontEdit;

    QMenu *fontMenu;
    QToolBar *fontToolBar;
    QComboBox *sizeComboBox;
    QAction *fontColorAct;

    uint fontSize;
    bool bShowStyles;
    QColor fontColor;
};
#endif
```



将字体全部显示的函数如下所示:

```
void MainWindow::listAllFonts()
{
    QFont font;
    QTextCharFormat textFormat;
    QFontDatabase database;
    QTextCursor cursor;

    textFormat.setForeground(fontColor);
    fontEdit->clear();
    foreach (QString family, database.families()) {
        const QStringList styles = database.styles(family);
        foreach (QString style, styles) {
            font = QFont(family, fontSize, database.weight(family, style),
                        database.italic(family, style));
            textFormat.setFont(font);
            cursor = fontEdit->textCursor();
            cursor.insertText (QString("%1 %2").arg(family).arg(style),
                            textFormat);
            cursor.insertBlock();
        }
    }
}
```

这个函数中使用了 `QFontDatabase` 类取得所有的系统字体, 然后再用 `foreach` 循环查询每种字体的风格。字体的风格使用 `QFontDatabase` 类的 `styles()` 函数查询, 可能的风格有“**Bold**”, “*Light Italic*”, “*Oblique*”等。这样得到所有的字体及其风格, 建立相应的字符在 `QTextEdit` 控件中显示。

另外还设置了改变显示字体的大小和颜色的菜单。相应的槽函数如下:

```
void MainWindow::changeFontColor()
{
    QColor color = QColorDialog::getColor(fontColor, this);
    if (color.isValid()) {
        fontColor = color;
        updateColor();
        listAllFonts();
    }
}

void MainWindow::fontSizeChanged(const QString &text)
{
    fontSize = text.toUInt();
    if (fontSize > 0)
        listAllFonts();
}
```

设置字体大小的方法有两种, 分别是 `setPointSize()` 和 `setPixelSize()`。其中 `setPixelSize()` 是设备相关的, `setPointSize()` 是设备无关的。

最后给出主窗口初始化的代码。

```
MainWindow::MainWindow(QWidget *parent)
: QMainWindow(parent)
{
    fontColor = Qt::black;
    fontSize = 16;

    createActions();
    createToolBars();
    createMenus();
    updateColor();

    fontEdit = new QTextEdit();
    setCentralWidget(fontEdit);

    resize(800,600);
    listAllFonts();
}

void MainWindow::createActions()
{
    fontColorAct = new QAction(QPixmap(32, 32), tr("Change Color..."), this);
    connect(fontColorAct, SIGNAL(triggered()), this,
            SLOT(changeFontColor()));
}

void MainWindow::createMenus()
{
    fontMenu = menuBar()->addMenu(tr("&Font"));
    fontMenu->addAction(fontColorAct);
}

void MainWindow::createToolBars()
{
    fontToolBar = addToolBar(tr("FontColor"));
    fontToolBar->addAction(fontColorAct);
    fontToolBar->addSeparator();
    QStringList fontSizes;
    fontSizes << "32" << "24" << "16" << "14" << "12" << "8" << "4" << "2";
    sizeComboBox = new QComboBox();
    sizeComboBox->setMinimumWidth(100);
    sizeComboBox->setEditable(true);
    sizeComboBox->addItem(fontSizes);
    sizeComboBox->setValidator(new QIntValidator(1, 65535, this));
    sizeComboBox->setCurrentIndex(2);
    fontToolBar->addWidget(sizeComboBox);
    connect(sizeComboBox, SIGNAL(editTextChanged(const QString&)), this,
```



```

        SLOT(fontSizeChanged(const QString&));
    }

void MainWindow::updateColor()
{
    QPixmap pixmap(32, 32);
    QPainter painter(&pixmap);
    painter.fillRect(0, 0, 32, 32, fontColor);

    QColor lighter = fontColor.light();
    painter.setPen(lighter);
    QPoint lightFrame[] = { QPoint(0, 31), QPoint(0, 0), QPoint(31, 0) };
    painter.drawPolyline(lightFrame, 3);

    painter.setPen(fontColor.dark());
    QPoint darkFrame[] = { QPoint(1, 31), QPoint(31, 31), QPoint(31, 1) };
    painter.drawPolyline(darkFrame, 3);
    painter.end();

    fontColorAct->setIcon(pixmap);
}

```

updateColor()是生成当前颜色工具按钮的函数，程序运行效果如图 6-12 所示。

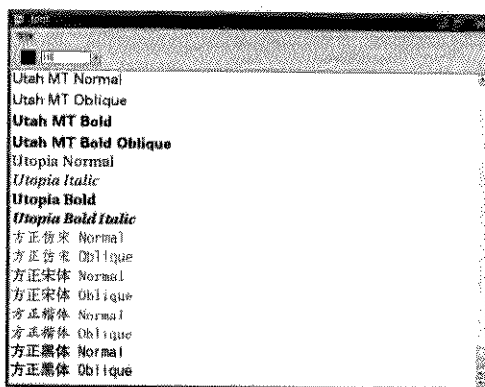


图 6-12 系统字体显示

6.4 绘图路径——QPainterPath

绘图路径 (painter path) 由基本图元 (矩形、椭圆、直线和曲线等) 组成。绘图路径可以是闭合的路径，如矩形和圆；或者是非闭合的路径，如直线和曲线。绘图路径在 Qt 中使用 QPainterPath 类表示，它提供了绘图操作的容器，可以使图形能够复用。

绘图路径可以进行填充、显示轮廓和剪裁。要生成可填充的轮廓的绘图路径，可以使用 QPainterPathStroker 类。使用 QPainterPath 的优点是复杂图形只需创建一次，就可以多次使用。

QPainterPath 对象可以是只有起点的空路径,或者从其他 QPainterPath 对象复制。创建了 QPainterPath 对象后,可以使用 moveTo()、arcTo()、cubicTo()、quadTo()函数将直线和曲线添加到路径中来。直线和曲线从 currentPosition()开始绘制。currentPosition()总是返回最后的子路径绘制的终点。使用 moveTo()函数可以在不增加路径的情况下移动 currentPosition(),它关闭上一个子路径,开始一个新的子路径。closeSubPath()也可以关闭当前的路径,并从 currentPosition()连接一条直线到绘图路径的起点。

QPainter 可以使用 addEllipse()、addPath()、addRect()、addRegion()和 addText()将 Qt 的一些基本图元加入绘图路径。一个已有的绘图路径可以通过 connectPath()函数加入到另一个绘图路径中。

如下代码使用 QPainterPath 的绘制了一个箭头。

```
QPainterPath path;
path.moveTo(10,100);
path.cubicTo(10, 100, 100, 10, 200, 70);
path.lineTo(200, 50);
path.lineTo(220, 80);
path.lineTo(200, 110);
path.lineTo(200, 90);
path.cubicTo(200, 100, 100, 50, 50, 100);
```

```
QPainter painter(this);
QPen pen(QColor(255, 0, 0), 2);
painter.setPen(pen);
painter.drawPath(path);
```

生成的图像如图 6-13 所示



图 6-13 QPainterPath 生成的箭头

Qt 提供了两种路径填充方式: Qt::OddEvenFill 和 Qt::WindingFill。Qt::OddEven 是默认的填充规则,它指定 QPainterPath 使用奇偶填充规则。该规则判断一个点是否在路径图形内的方法是从该点画一条水平线到路径图形外,计算水平线和路径的交点数,如果交点是奇数个则说明该点在路径图形内。

QPainterPath 还有一些函数可以获取路径的信息,如 elementAt()函数可以取出指定的子路径元素,isEmpty()函数判断当前路径是否为空,controlPointRect()函数返回路径中所有的点和控制点的矩形,该函数运行速度比返回精确包容框的 boundingRect()函数快得多。contains()函数判断一个点或矩形是否在路径内,intersects()判断指定的矩形与路径是否相交。

QPainterPath 可以将矩形图形转换为其他的图形,如使用 toFillPolygon(), toFillPolygons()和 toSubpathPolygons()函数将路径转换为多边形。

QPainterPath 还可以使用文字作为路径。下面的代码演示了文字路径,并使用线性渐变填充。

```
QLinearGradient linearGrad(QPointF(200, 0), QPointF(1000, 0));
linearGrad.setColorAt(0, Qt::black);
linearGrad.setColorAt(1, Qt::white);
QFont font("隶书", 80);
```



```
font.SetBold(true);
QPainterPath textPath;
textPath.addText(200, 300, font, tr("电子工业出版社"));
painter.setBrush(linearGrad);
painter.drawPath(textPath);
```

程序运行效果如图 6-14 所示。

电子工业出版社

图 6-14 文字组成的绘图路径

6.5 QImage 与 QPixmap 绘图设备

6.5.1 QImage

Qt 提供了 4 个处理图像的类：QImage、QPixmap、QBitmap 和 QPicture。它们有着各自的特点，QImage 优化了 I/O 操作，可以直接存取操纵像素数据；QPixmap 主要用来在屏幕上显示图像；QBitmap 从 QPixmap 继承，只能表示两种颜色；QPicture 是可以记录和重放 QPainter 命令的类。

QImage 类提供了与硬件无关的图像表示方法，通过 QImage 可以直接存取像素数据，QImage 也可以用作绘图设备。QImage 支持的图像颜色可以是单色、8 位、32 位和 Alpha 混合格式。

因为 QImage 从 QPaintDevice 继承，所以 QPainter 可以直接在 QImage 上绘图。除了绘制文字外（QFont 依赖于底层 GUI），其他的绘制操作可以在任意线程中完成。如果要在其他线程中绘制文字，可以使用 QPainterPath。QImage 对象具有隐式共享（在第 13 章中会详细介绍），作为传值参数，可以使用数据流及进行比较等特性。

读入图像文件数据可以通过 QImage 的构造函数、load() 函数、loadFromData() 几种方法完成。还可以通过 QImage 的静态函数 fromData() 由指定数据构造一个 QImage 对象。既可以从文件系统装入，也可以从 Qt 应用程序的嵌入式资源中读取。使用 save() 函数可以保存 QImage 对象。

可以通过 QImageReader::supportedImageFormats() 和 QImageWriter::supportedImageFormats() 函数获取 QImage 支持的所有文件格式列表。Qt 支持如表 6-3 所示格式的图像文件。

表 6-3 QImage 支持的图像文件

格 式	支持能力
BMP	读写
GIF	读
JPG	读写
JPEG	读写
PNG	读写
PBM	读
PGM	读
PPM	读写
XBM	读写
XPM	读写

如果要支持新的图像格式，可以编写相应的插件并加载。QImage 的函数很丰富，可以分为如表 6-4 所示的几类。

表 6-4 QImage 函数概览

类 别	函 数
几何函数	size().width().height().dotsPerMeterX().dotsPerMeterY()函数获取图像大小和比例信息 rect()函数返回图像的包容矩形，valid()函数测试给定的坐标是否在此矩形内，offset()函数获取图像和其他图像之间的相对偏移量，setOffset()函数设置偏移量
颜色函数	某个像素的颜色可以通过 pixel()函数获取，返回值是 QRgb 类型，对于单色和 256 色图像，colorTable()函数返回调色板，numColors()返回调色板中的条目数，用 pixelIndex()函数获取像素的颜色索引，然后使用 color()函数取出实际的颜色值 hasAlphaChannel()函数返回图像是否使用了 alpha 通道，allGray()和 isGrayscale()函数测试图像是否为灰度图像
文字	text()函数返回图像附属的文字，textKeys()返回文字键值列表，setText()函数改变图像附属的文字
低级信息	depth()获取图像颜色位数。支持的位数是 1、8 和 32 位 format().bytesPerLine()和 numBytes()函数返回图像的数据存储信息 serialNumber()函数取得唯一标识 QImage 对象的数字

QImage 的 8 位和单色图像采用颜色索引表的方式存取，32 位的图像则直接存储 ARGB 值，因此它们像素的操作函数也不相同。对 32 位图像，setPixel()函数可以改变指定像素的 QRgb 颜色值。对 8 位和单色图像，setPixel()改变在预定义颜色表中的索引值。如果要改变颜色表，可以使用 setColor()函数。

QImage 提供 scanLine()函数返回指定行的数据，bits()函数返回第一个像素的指针。每个像素在 QImage 中都使用整数表示。单色图像使用一位的索引指向只有两种颜色的调色板。有两种类型的单色图像，big endian(MSB)和 little endian(LSB)。

256 色的图像使用 8 位索引的调色板，调色板的数据类型是 QVector<QRgb>，QRgb 实际上是无符号整型数，存储 ARGB 的格式是 0xAARRGGBB。32 位的图像直接存储。有三种类型的存储格式：RGB，ARGB 和已预乘 (premultiplied) 的 ARGB 值。在已预乘 ARGB 类型中，红绿蓝三色已经和 alpha 相乘并模除 255。

allGray()和 isGrayscale()函数可以判断一个彩色图像能否安全地转化为灰度图像。图像的格式可以用 format()函数取出，convertToFormat()可以进行图像格式转换，QImage 支持的存储格式如表 6-5 所示。

表 6-5 QImage 图像存储格式

格式常量	说 明
QImage::Format_Mono	单色图像 (MSB 方式)
QImage::Format_MonoLSB	单色图像 (LSB 方式)
QImage::Format_Indexed8	使用颜色表的 256 色图像
QImage::Format_RGB32	不支持 Alpha 通道的 32 位图像
QImage::Format_ARGB32	含 Alpha 通道的 32 位图像
QImage::Format_ARGB32_Premultiplied	已预乘的含 Alpha 通道的 32 位图像

6.5.2 QPixmap

QPixmap 主要完成屏幕后台 (off-screen) 缓冲区绘图。QPixmap 对象可以使用 QLabel 或 QAbstractButton 子类 (QPushButton 和 QToolButton) 显示。QLabel 通过设置 pixmap 属性，



QAbstractButton 通过设置 icon 属性来完成。

除了使用构造函数来初始化, QPixmap 对象还可以使用静态函数 grabWidget() 和 grabWindow() 函数创建, 并绘制指定的窗口和窗口部件。

QPixmap 中的像素数据是内部的, 并且由底层的窗口系统进行管理。如果要存取像素, 只有通过 QPainter 函数或将 QPixmap 对象转换为 QImage 对象。根据底层系统的不同, QPixmap 可以 RGB32 或混合 alpha 格式存储。如果图像有 Alpha 通道且底层系统允许, 则优先使用混合 alpha 格式。因此 QPixmap 是依赖于底层系统的。在 X11 和 Mac 系统上, QPixmap 存储在服务器端, QImage 存储在客户端。在 Windows 系统上, 这两个类是用相同方式表示的。

QImage 和 QPixmap 可以相互转换。通常 QImage 载入图像并进行直接操作, 然后转换为 QPixmap 在屏幕上显示。如果不需要操作像素, 就直接使用 QPixmap。在 Windows 上, QPixmap 还可以与 HBITMAP 之间互相转换。QPixmap 同 QImage 一样也使用隐式共享, 也能够使用数据流。

下面通过实现图像浏览器来学习在 Qt 中如何处理图像。这个图像浏览器能够实现图像的浏览、旋转和放大等简单功能。

用来显示图像的窗口部件定义如下:

```
#ifndef IMAGEWIDGET_H_
#define IMAGEWIDGET_H_

#include <QtGui>

class ImageWidget : public QWidget
{
    Q_OBJECT

public:
    bool bFit;           // 图像是否匹配窗口尺寸
    qreal scale;         // 图像缩放值

    ImageWidget(QWidget *parent = 0);
    void setPixmap(QString fileName);
    QPixmap getPixmap();
    void setAngle(qreal rotateAngle);

protected:
    void paintEvent(QPaintEvent *event);

private:
    QPixmap pixmap;
    qreal angle;
};

#endif /*IMAGEWIDGET_H_*/
```

ImageWidget 类定义中的 setAngle() 函数用来设置图像旋转的角度。

ImageWidget 类实现文件如下:

```
#include <QtCore>
#include "imagewidget.h"
```



```
ImageWidget::ImageWidget(QWidget *parent)
: QWidget(parent)
{
    QDesktopWidget desktop;
    pixmap = QPixmap(desktop.width(), desktop.height());
    scale = 1;
    angle = 0;
    bFit = true;
}
```

在构造函数中设置了变量的初始值。

绘图函数 `paintEvent()` 完成图像的显示。

```
void ImageWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    if(angle)
    {
        QPointF center(width()/2.0,height()/2.0);
        painter.translate(center);
        painter.rotate(angle);
        painter.translate(-center);
    }
    if(bFit)
    {
        QPixmap fitPixmap = pixmap.scaled(width(),height(),
            Qt::KeepAspectRatio);
        painter.drawPixmap(0, 0, fitPixmap);
    }
    else
        painter.drawPixmap(0, 0, pixmap);
}
```

在绘图函数中判断图像是否要旋转，是否要按实际大小显示。

设置和获取当前显示的图像的函数 `setPixmap()` 和 `getPixmap()`：

```
void ImageWidget::setPixmap(QString fileName)
{
    pixmap.load(fileName);
    update();
}

QPixmap ImageWidget::getPixmap()
{
    return pixmap;
}
```

设置旋转角度的函数 `setAngle()`：

```
void ImageWidget::setAngle(qreal rotateAngle)
{
}
```



```
    angle += rotateAngle;
    update();
}
```

主窗口的头文件如下:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QScrollArea>
#include <QDir>
#include "imagewidget.h"

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();

public slots:
    void selectDir();
    void next();
    void prev();
    void rotateLeft();
    void rotateRight();
    void zoomIn();
    void zoomOut();
    void actualSize();
    void fitSize();
    void copy();
    void print();
    void present();

protected:
    void resizeEvent(QResizeEvent * event);

private:
    void createActions();
    void createMenus();
    void createToolBars();
    void createStatusBar();

    QScrollArea *scrollArea;
    ImageWidget *imageWidget;

    QMenu *naviMenu;
    QMenu *operMenu;

    QToolBar *naviToolBar;
```

```

QToolBar *operToolBar;

QAction *dirAct;
QAction *nextAct;
QAction *prevAct;
QAction *leftAct;
QAction *rightAct;
QAction *zoomInAct;
QAction *zoomOutAct;
QAction *actualSizeAct;
QAction *fitSizeAct;
.....
QStringList imageList;
int index;
QDir imageDir;
QClipboard *clipboard;
};
#endif

```

主要在头文件中定义了菜单、按钮及对应的 QAction。

主窗口的实现文件如下：

```

#include <QtGui>
#include "mainwindow.h"

MainWindow::MainWindow()
{
    imageWidget = new ImageWidget;
    scrollArea = new QScrollArea;
    scrollArea->setBackgroundRole(QPalette::Dark);
    imageWidget->setSizePolicy(QSizePolicy::Ignored,
        QSizePolicy::Ignored);
    scrollArea->setWidget(imageWidget);
    scrollArea->widget()->setMinimumSize(320, 240);
    setCentralWidget(scrollArea);

    createActions();
    createMenus();
    createToolBars();
    createStatusBar();
    setWindowTitle(tr("zeki"));
    setFocusPolicy(Qt::StrongFocus);

    index = 0;

    imageDir.setPath("/windows/E/Wife/USA photo/5.3");
    QStringList filter;
    filter << "*.jpg" << "*.bmp" << "*.jpeg" << "*.png" << "*.xpm";
    imageList = imageDir.entryList ( filter, QDir::Files );
    next();
}

```



```
void MainWindow::resizeEvent(QResizeEvent * event)
{
    QRect childRect = scrollArea->childrenRect();
    imageWidget->resize(childRect.size());
}

void MainWindow::createActions()
{
    dirAct = new QAction(QIcon(":/images/open.png"), tr("Open"), this);
    dirAct->setShortcut(QKeySequence::Open);
    connect(dirAct, SIGNAL(triggered()), this, SLOT(selectDir()));

    prevAct = new QAction(QIcon(":/images/previous.png"),
        tr("Previous"), this);
    prevAct->setShortcut(QKeySequence::Back);
    connect(prevAct, SIGNAL(triggered()), this, SLOT(prev()));

    nextAct = new QAction(QIcon(":/images/next.png"), tr("Next"), this);
    nextAct->setShortcut(QKeySequence::Forward);
    connect(nextAct, SIGNAL(triggered()), this, SLOT(next()));

    leftAct = new QAction(QIcon(":/images/rotate_left.png"),
        tr("Left"), this);
    leftAct->setShortcut(tr("Ctrl+L"));
    connect(leftAct, SIGNAL(triggered()), this, SLOT(rotateLeft()));

    rightAct = new QAction(QIcon(":/images/rotate_right.png"),
        tr("Right"), this);
    rightAct->setShortcut(tr("Ctrl+R"));
    connect(rightAct, SIGNAL(triggered()), this, SLOT(rotateRight()));

    zoomInAct = new QAction(QIcon(":/images/zoomin.png"),
        tr("ZoomIn"), this);
    zoomInAct->setShortcut(QKeySequence::ZoomIn);

    connect(zoomInAct, SIGNAL(triggered()), this, SLOT(zoomIn()));
    zoomOutAct = new QAction(QIcon(":/images/zoomout.png"),
        tr("ZoomOut"), this);
    zoomOutAct->setShortcut(QKeySequence::ZoomOut);
    connect(zoomOutAct, SIGNAL(triggered()), this, SLOT(zoomOut()));

    actualSizeAct = new QAction(QIcon(":/images/actualsize.png"),
        tr("Actual"), this);
    actualSizeAct->setShortcut(Qt::Key_Home);
    connect(actualSizeAct, SIGNAL(triggered()), this, SLOT(actualSize()));

    fitSizeAct = new QAction(QIcon(":/images/fitwindow.png"),
        tr("Fit"), this);
    fitSizeAct->setShortcut(Qt::Key_End);
```

```

        connect(fitSizeAct, SIGNAL(triggered()), this, SLOT(fitSize()));
        .....
    }

void MainWindow::createMenus()
{
    naviMenu = menuBar()->addMenu(tr("Navigation"));
    naviMenu->addAction(prevAct);
    naviMenu->addAction(nextAct);

    operMenu = menuBar()->addMenu(tr("Operation"));
    operMenu->addAction(leftAct);
    operMenu->addAction(rightAct);
    operMenu->addAction(zoomInAct);
    operMenu->addAction(zoomOutAct);
    operMenu->addAction(actualSizeAct);
    operMenu->addAction(fitSizeAct);
    .....
}

void MainWindow::createToolBars()
{
    naviToolBar = addToolBar(tr("Navigation"));
    naviToolBar->addAction(dirAct);
    naviToolBar->addSeparator();
    naviToolBar->addAction(prevAct);
    naviToolBar->addAction(nextAct);

    operToolBar = addToolBar(tr("Operation"));
    operToolBar->addAction(leftAct);
    operToolBar->addAction(rightAct);
    operToolBar->addAction(zoomInAct);
    operToolBar->addAction(zoomOutAct);
    operToolBar->addAction(actualSizeAct);
    operToolBar->addAction(fitSizeAct);
    .....
}

void MainWindow::createStatusBar()
{
    statusBar()->showMessage(tr("Ready"));
}

// 选择浏览的目录
void MainWindow::selectDir()
{
    QString dir = QFileDialog::getExistingDirectory(this,
        tr("Open Directory"), QDir::currentPath(),
        QFileDialog::ShowDirsOnly | QFileDialog::DontResolveSymlinks);
    if(dir.isEmpty())
        return;
}

```



```
imageDir.setPath(dir);
QStringList filter;
filter << "*.jpg" << "*.bmp" << "*.jpeg" << "*.png" << "*.xpm";
imageList = imageDir.entryList ( filter, QDir::Files );
next();
}
// 下一幅图像
void MainWindow::next()
{
    if(index < imageList.size())
    {
        imageWidget->setPixmap(imageDir.absolutePath() + QDir::separator()
            + imageList.at(index));
        statusBar()->showMessage(imageList.at(index));
        index++;
    }
}
// 前一幅图像
void MainWindow::prev()
{
    if(index > 0)
    {
        imageWidget->setPixmap(imageDir.absolutePath() + QDir::separator()
            + imageList.at(index));
        statusBar()->showMessage(imageList.at(index));
        index--;
    }
}
// 左转 90°
void MainWindow::rotateLeft()
{
    imageWidget->setAngle(-90);
}
// 右转 90°
void MainWindow::rotateRight()
{
    imageWidget->setAngle(90);
}
// 放大图像
void MainWindow::zoomIn()
{
    imageWidget->scale *= 1.25;
    zoomInAct->setEnabled(imageWidget->scale < 3);
    zoomOutAct->setEnabled(imageWidget->scale > 0.333);
    imageWidget->resize(imageWidget->scale * scrollArea->size());
}
// 缩小图像
void MainWindow::zoomOut()
{
    imageWidget->scale *= 0.8;
```

```

        zoomInAct->setEnabled(imageWidget->scale < 3);
        imageWidget->re_size(imageWidget->scale * scrollArea->size());
    }
    // 显示图像实际大小
    void MainWindow::actualSize()
    {
        imageWidget->scale = 1;
        imageWidget->bFit = false;
        imageWidget->update();
    }
    // 匹配窗口大小
    void MainWindow::fitSize()
    {
        imageWidget->scale = 1;
        imageWidget->bFit = true;
        imageWidget->update();
    }
    // 全屏显示
    void MainWindow::present()
    {
        statusBar()->hide();
        menuBar()->hide();
        naviToolBar->hide();
        operToolBar->hide();
        showFullScreen();
    }
}

```

程序的运行效果如图 6-15 所示。

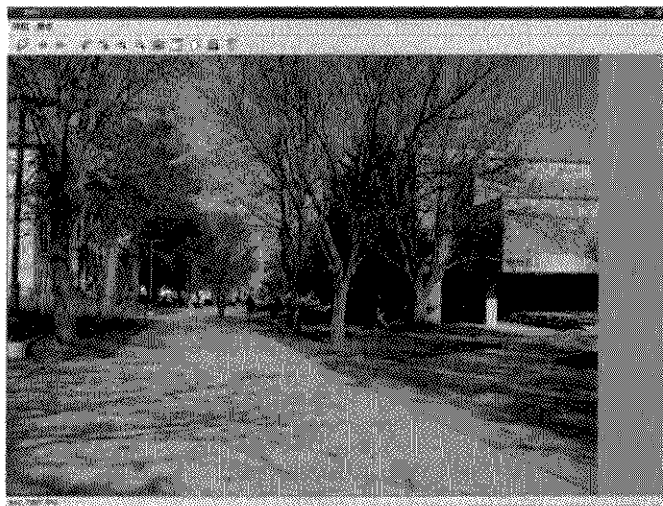


图 6-15 图像浏览程序

6.6 组合模式绘图

组合模式 (Composition Mode) 用来指定如何合并源图像和一个图像。最常见的是 SourceOver (通常也叫 Alpha 混合), 当源像素和目标像素以这种方式混合时, 源图像的 Alpha 通道定义了像素的透明度。

组合模式绘图只支持 Format_ARGB32_Premultiplied 和 Format_ARGB32 格式, 而且应该优先使用 Format_ARGB32_Premultiplied 格式。设置了组合模式后, 它对所有的绘图操作都有效, 如画笔、画刷、渐变效果和 pixmap/image 绘制。

QPainter::CompositionMode 枚举类型中前 12 种组合类型是 T.Porter 和 T.Duff 于 1984 年在论文《Compositing Digital Images》中阐明的 12 条混合规则 (Porter-Duff 规则)。混合的计算方法在此给出, 以便理解混合的过程。

首先定义混合因子:

- A_s : 源像素的 alpha 分量
- C_s : 源像素中计算好的 (premultiplied) 色彩分量
- A_d : 目标像素的 alpha 分量
- C_d : 目标像素中计算好的色彩分量
- F_s : 源像素在输出结果中占有的比例
- F_d : 目标像素在输出结果中占有的比例
- A_r : 输出结果中的 Alpha 分量
- C_r : 输出结果中的计算好的色彩分量

Porter 和 Duff 定义了选择混合因子 F_s 和 F_d 产生不同的视觉效果 12 种规则。最终结果中的 Alpha 值和色彩由下面的公式决定。

$$\begin{aligned} F_s &= f(A_d) \\ F_d &= f(A_s) \\ A_r &= A_s \times F_s + A_d \times F_d \\ C_r &= C_s \times F_s + C_d \times F_d \end{aligned}$$

每种类型 F_s 和 F_d 取值如表 6-6 所示。

表 6-6 Porter-Duff 规则

常 量	F_s	F_d	说 明
QPainter::CompositionMode_SourceOver	1	$1-A_d$	默认模式, 源 Alpha 和目标像素混合
QPainter::CompositionMode_DestinationOver	$1-A_d$	1	和 source over 相反, 目标 Alpha 和源像素混合
QPainter::CompositionMode_Clear	N/A	N/A	清除目标像素
QPainter::CompositionMode_Source	N/A	N/A	输出源像素
QPainter::CompositionMode_Destination	N/A	N/A	输出目标像素
QPainter::CompositionMode_SourceIn	A_d	0	在目标部分的源替代目标
QPainter::CompositionMode_DestinationIn	0	A_s	和 source in 相反
QPainter::CompositionMode_SourceOut	$1-A_d$	0	在目标之外的源替代目标
QPainter::CompositionMode_DestinationOut	0	$1-A_s$	和 source out 相反
QPainter::CompositionMode_SourceAtop	A_d	$1-A_s$	在目标部分的源和目标组合
QPainter::CompositionMode_DestinationAtop	$1-A_d$	A_s	和 source atop 相反
QPainter::CompositionMode_Xor	$1-A_d$	$1-A_s$	目标之外的源和源之外的目标混合

需要注意,上面的说明并没有完全概括各种混合的含义,要准确理解它们可以看公式并进行实践。除了上面 12 条 Porter-Duff 规则外,Qt 还支持 12 种扩展混合模式。下面给出计算公式,需要注意如果计算结果中 alpha 值和色彩值超过 0~255 的范围,数值将会被截断。

1. QPainter::CompositionMode_Plus

源和目标相加。该操作可以实现动画中两幅图像的溶解过渡效果。

$$\begin{aligned}C_r &= C_s + C_d \\A_r &= A_s + A_d\end{aligned}$$

2. QPainter::CompositionMode_Multiply

源和目标进行正片叠底 (multiply) 操作。结果的颜色至少是源和目标中较暗的颜色。任何颜色和黑色作该操作产生黑色,任何颜色和白色进行正片叠底操作将不会改变。

$$\begin{aligned}C_r &= C_s \times C_d + C_s \times (1 - A_d) + C_d \times (1 - A_s) \\A_r &= A_s \times A_d + A_s \times (1 - A_d) + A_d \times (1 - A_s) = A_s + A_d - A_s \times A_d\end{aligned}$$

3. QPainter::CompositionMode_Screen

源和目标互补然后相乘。结果的颜色至少是源和目标中较亮的颜色。因为任何颜色和黑色进行滤色 (screen) 操作不会改变,任何颜色和白色进行滤色操作还是白色。

$$\begin{aligned}C_r &= C_s + C_d - C_s \times C_d \\A_r &= A_s + A_d - A_s \times A_d\end{aligned}$$

4. QPainter::CompositionMode_Overlay

根据目标颜色值的不同,进行相乘操作或滤色操作。源色彩保持亮度和阴影覆盖在目标上。目标颜色和源颜色混合以反映目标的亮度。

$$\begin{aligned}&\text{if}(2 \times C_d < A_d) \\&\quad C_r = 2 \times C_s \times C_d + C_s \times (1 - A_d) + C_d \times (1 - A_s) \\&\quad \text{else} \\&\quad C_r = A_s \times A_d - 2 \times (A_d - C_d) \times (A_s - C_s) + C_s \times (1 - A_d) + C_d \times (1 - A_s) \\&\quad A_r = A_s + A_d - A_s \times A_d\end{aligned}$$

5. QPainter::CompositionMode_Darken

选择源和目标中较暗的颜色。

$$\begin{aligned}&\text{if}(C_s \times A_d < C_d \times A_s) \\&\quad \text{src_over}() \\&\quad \text{else} \\&\quad \text{dst_over}()\end{aligned}$$

6. QPainter::CompositionMode_Lighten

选择源和目标中较亮的颜色。

$$\begin{aligned}&\text{if}(C_s \times A_d > C_d \times A_s) \\&\quad \text{src_over}()\end{aligned}$$



```
else
    dst_over()
```

7. QPainter::CompositionMode_ColorDodge

加亮目标颜色以反映源色彩。绘制黑色将没有效果。

```
if ( $C_s \times A_d + C_d \times A_s > A_s \times A_d$ )
     $C_r = A_s \times A_d + C_s \times (1 - A_d) + C_d \times (1 - A_s)$ 
else
     $C_r = C_d \times A_s / (1 - C_s / A_s) + C_s \times (1 - A_d) + C_d \times (1 - A_s)$ 
 $A_r = A_s + A_d - A_s \times A_d$ 
```

8. QPainter::CompositionMode_ColorBurn

使目标颜色变暗以反映源色彩。绘制白色将没有效果。

```
if ( $C_s \times A_d + C_d \times A_s \leq A_s \times A_d$ )
     $C_r = C_s \times (1 - A_d) + C_d \times (1 - A_s)$ 
else
     $C_r = A_s \times (C_s \times A_d + C_d \times A_d - A_s \times A_d) / C_s + C_s \times (1 - A_d) + C_d \times (1 - A_s)$ 
 $A_r = A_s + A_d - A_s \times A_d$ 
```

9. QPainter::CompositionMode_HardLight

根据源的颜色，决定是正片叠底还是滤色操作。如果源颜色亮度值高于 0.5（亮度取值 0-1），目标将变亮，即进行了滤色操作。如果源颜色亮度值低于 0.5，目标将会变暗，相当于进行了正片叠底操作。如果源亮度值等于 0.5，目标不发生改变。变亮或变暗的程度取决于源色彩和 0.5 的差值。绘制纯黑或纯白结果还是纯黑或纯白。

```
if ( $2 \times C_s < A_s$ )
     $C_r = 2 \times C_s \times C_d + C_s \times (1 - A_d) + C_d \times (1 - A_s)$ 
else
     $C_r = A_s \times A_d - 2 \times (A_d - C_d) \times (A_s - C_s) + C_s \times (1 - A_d) + C_d \times (1 - A_s)$ 
 $A_r = A_s + A_d - A_s \times A_d$ 
```

10. QPainter::CompositionMode_SoftLight

根据源的颜色，决定进行变暗（darken）操作还是变亮（lighten）操作。如果源颜色比 0.5 亮，目标将变亮，即进行了滤色操作。如果源颜色比 0.5 暗，目标将会变暗，相当于进行了颜色加深（burn）操作。如果等于 0.5，目标不发生改变。变亮或变暗的程度取决于源色彩和 0.5 的差值。

```
if ( $2 \times C_s < A_s$ )
     $C_r = C_d \times (A_s - (1 - C_d / A_d) \times (2 \times C_s - A_s)) + C_s \times (1 - A_d) + C_d \times (1 - A_s)$ 
elseif ( $8 \times C_d \leq A_d$ )
     $C_r = C_d \times (A_s - (1 - C_d / A_d) \times (2 \times C_s - A_s) \times (3 - 8 \times C_d / A_d)) + C_s \times (1 - A_d) + C_d \times (1 - A_s)$ 
else
     $C_r = (C_d \times A_s + ((C_d / A_d)^{0.5} \times A_d - C_d) \times (2 \times C_s - A_s)) + C_s \times (1 - A_d) + C_d \times (1 - A_s)$ 
 $A_r = A_s + A_d - A_s \times A_d$ 
```

11. QPainter::CompositionMode_Difference

源和目标中较暗的颜色减去较亮的颜色。绘制白色导致反转目标颜色，绘制黑色没有任何变化。

$$C_r = C_s + C_d - 2 \times \min(C_s \times A_d, C_d \times A_s)$$

$$A_r = A_s + A_d - A_s \times A_d$$

12. QPainter::CompositionMode_Exclusion

和上一条规则的效果类似，但对对比度要低一些。绘制白色导致反转目标颜色，绘制黑色没有任何变化。

$$C_r = (C_s \times A_d + C_d \times A_s - 2 \times C_s \times C_d) + C_s \times (1 - A_d) + C_d \times (1 - A_s)$$

$$A_r = A_s + A_d - A_s \times A_d$$

下面通过一个例子来观察不同模式的绘图效果。例子中目标图像使用的图像文件中，源图形使用一个可调颜色和 Alpha 值的矩形。用户可以自由选择组合模式和源图像以便观察混合效果。

用来绘图的窗口部件定义如下：

```
class CompositionCanvas : public QWidget
{
    Q_OBJECT

public:
    CompositionCanvas(QWidget *parent = 0, Qt::WindowFlags f = 0);

public slots:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void compositeModeChanged(int currentRow);
    void alphaChanged(int value);
    void redChanged(int value);
    void greenChanged(int value);
    void blueChanged(int value);

private:
    QPoint pos;
    QImage srcImage;
    QImage dstImage;
    QImage resultImage;
    QPainter::CompositionMode currentMode;
    QColor color;
};
```

这里定义了三个 QImage 对象：srcImage、dstImage 和 resultImage，分别表示源图像、目标图像和结果图像。pos 表示目标图像绘制的左上角位置。定义了 mousePressEvent() 响应鼠标事件，根据鼠标点击的位置移动目标图像位置。compositeModeChanged() 槽响应组合模式改变的信号并重绘。alphaChanged()、redChanged()、greenChanged() 和 blueChanged() 分别响应 Alpha 和红、绿、蓝三种颜色分量变化并重绘。

绘图窗口部件的构造函数实现如下：



```
CompositionCanvas::CompositionCanvas(QWidget *parent, Qt::WindowFlags f)
: QWidget(parent, f)
{
    pos.setX(400);
    pos.setY(550);
    setMinimumSize(700,700);
    srcImage = QImage(1000,1000, QImage::Format_ARGB32_Premultiplied);
    resultImage = QImage(1000,1000, QImage::Format_ARGB32_Premultiplied);
    srcImage.load("build.jpg");
    dstImage = QImage(100,100, QImage::Format_ARGB32_Premultiplied);
    currentMode = QPainter::CompositionMode_SourceOver;
    color = QColor(255,0,0,255);
}
```

在构造函数中对 `srcImage`、`dstImage` 和 `resultImage` 进行了初始化。同时还初始化了一些控制变量。接下来的 `paintEvent()` 事件是运用了组合模式绘图，代码如下：

```
void CompositionCanvas::paintEvent(QPaintEvent *event)
{
    QPainter painter(&resultImage);
    painter.fillRect(resultImage.rect(), Qt::transparent);
    painter.drawImage(0, 0, srcImage);
    painter.setCompositionMode(currentMode);
    dstImage.fill(color.rgb());
    painter.drawImage(pos, dstImage);

    QPainter p(this);
    p.drawImage(0,0,resultImage);
}
```

`paintEvent()` 中使用了当前的组合模式将两个 `QImage` 对象混合后在屏幕上显示。这是因为组合模式只能在 `QImage` 绘图设备上进行绘制，而不能直接在 `QWidget` 上绘制，所以程序在 `QImage` 对象上绘制好图形后再贴到 `QWidget`。

下面的函数对组合模式、Alpha 值、红、绿、蓝分量改变进行响应。

```
void CompositionCanvas::compositeModeChanged(int currentRow)
{
    currentMode = (QPainter::CompositionMode)currentRow;
    update();
}

void CompositionCanvas::alphaChanged(int value)
{
    color.setAlpha(value);
    update();
}

void CompositionCanvas::redChanged(int value)
{
}
```

```

        color.setRed(value);
        update();
    }

void CompositionCanvas::greenChanged(int value)
{
    color.setGreen(value);
    update();
}

void CompositionCanvas::blueChanged(int value)
{
    color.setBlue(value);
    update();
}

void CompositionCanvas::mousePressEvent(QMouseEvent *event)
{
    pos = event->pos();
    update();
}

```

完成绘制代码后，还需要定义一个 `QWidget` 包含各种控件和绘图控件。定义如下：

```

class CompositionWidget : public QWidget
{
    Q_OBJECT

public:
    CompositionWidget(QWidget *parent = 0, Qt::WindowFlags f = 0);

private:
    QHBoxLayout *hLayout;
    QGridLayout *gridLayout;

    QLabel *aLabel;
    QLabel *rLabel;
    QLabel *gLabel;
    QLabel *bLabel;

    QSlider *aSlider;
    QSlider *rSlider;
    QSlider *gSlider;
    QSlider *bSlider;

    QListWidget *list;
    CompositionCanvas *canvas;
};

```

为了控制 `alpha`、红、绿、蓝颜色分量，定义了四个 `QSlider` 对象。使用了一个 `QListWidget` 来选择各种组合模式。对于 `CompositionWidget` 类，只需要实现其构造函数：

```
CompositionWidget::CompositionWidget(QWidget *parent, Qt::WindowFlags f)
: QWidget(parent, f)
{
```

```
    QStringList compositionList;
    compositionList << "Source Over" << "Destination Over" << "Clear"
        << "Source" << "Destination" << "Source In" << "Destination In"
        << "Source Out" << "Destination Out" << "Source Atop"
        << "Destination Atop" << "Xor" << "Plus" << "Multiply"
        << "Screen" << "Overlay" << "Darken" << "Lighten"
        << "ColorDodge" << "ColorBurn" << "HardLight"
        << "SoftLight" << "Difference" << "Exclusion";
```

```
    list = new QListWidget;
    list->addItems( compositionList );
    canvas = new CompositionCanvas;
```

```
    aLabel = new QLabel(tr("<b>Alpha:</b>"));
    rLabel = new QLabel(tr("<b>Red:</b>"));
    gLabel = new QLabel(tr("<b>Green:</b>"));
    bLabel = new QLabel(tr("<b>Blue:</b>"));
```

```
    aSlider = new QSlider;
    aSlider->setOrientation(Qt::Horizontal);
    aSlider->setMinimum(0);
    aSlider->setMaximum(255);
    aSlider->setValue(255);
```

```
    rSlider = new QSlider;
    rSlider->setOrientation(Qt::Horizontal);
    rSlider->setMinimum(0);
    rSlider->setMaximum(255);
    rSlider->setValue(255);
```

```
    gSlider = new QSlider;
    gSlider->setOrientation(Qt::Horizontal);
    gSlider->setMinimum(0);
    gSlider->setMaximum(255);
    gSlider->setValue(0);
```

```
    bSlider = new QSlider;
    bSlider->setOrientation(Qt::Horizontal);
    bSlider->setMinimum(0);
    bSlider->setMaximum(255);
    bSlider->setValue(0);
```

```
    gridLayout = new QGridLayout;
    gridLayout->addWidget(aLabel, 0, 0);
    gridLayout->addWidget(aSlider, 0, 1);
    gridLayout->addWidget(rLabel, 1, 0);
    gridLayout->addWidget(rSlider, 1, 1);
    gridLayout->addWidget(gLabel, 2, 0);
```

```

gridLayout->addWidget(gSlider, 2, 1);
gridLayout->addWidget(bLabel, 3, 0);
gridLayout->addWidget(bSlider, 3, 1);
gridLayout->addWidget(list, 4, 0, 15, 2);

hLayout = new QHBoxLayout;
hLayout->addWidget(canvas);
hLayout->addLayout(gridLayout);
setLayout(hLayout);

connect(list, SIGNAL(currentRowChanged(int)), canvas,
        SLOT(compositeModeChanged(int)));
connect(aSlider, SIGNAL(valueChanged(int)), canvas,
        SLOT(alphaChanged(int)));
connect(rSlider, SIGNAL(valueChanged(int)), canvas,
        SLOT(redChanged(int)));
connect(gSlider, SIGNAL(valueChanged(int)), canvas,
        SLOT(greenChanged(int)));
connect(bSlider, SIGNAL(valueChanged(int)), canvas,
        SLOT(blueChanged(int)));
}

```

构造函数生成了各种控件，并将 QSlider 对象的值限定在 0~255 之间。最后进行了信号和槽的连接。最后完成 main() 函数。

```

int main(int argc, char **argv)
{
    QApplication app(argc, argv);

    CompositionWidget compWidget;
    compWidget.show();

    return app.exec();
}

```

程序的运行界面如图 6-16 所示，图中的矩形处显示了叠加的效果。



图 6-16 组合模式绘图



用户可以改变各种模式和 ARGB 值来测试混合效果, 最好还能将背景图替换进行测试。

6.7 Graphics View 框架

Qt 4.2 开始引入 Graphics View 框架用来取代 Qt 3 中的 Canvas 模块, 并在很多地方做了改进。Graphics View 框架实现了模型—视图结构的图形管理, 能对大量的图元进行管理, 支持碰撞检测、坐标变换和图元组等多种方便的功能。

Graphics View 中增强的表现系统可以利用 Qt 4 绘图系统的反锯齿、OpenGL 工具来改善绘图性能。Graphics View 支持事件传播体系结构, 可以使图元在场景(scene)中得到提高了一倍的精确交互能力。图元能够处理键盘事件, 鼠标按下、移动、释放和双击事件, 也能跟踪鼠标的移动。在 Graphics View 框架中, 通过 BSP (二元空间划分树, Binary Space Partitioning) 来提供快速的图元查找, 这样就能实时地显示大场景, 甚至上百万个图元。

6.7.1 Graphics View 体系结构

Graphics View 框架提供基于图元的模型—视图编程, 类似于 Qt InterView (第 15 章详细讲解) 的模型—视图结构, 只是这里的数据是图形。Graphics View 框架中包括三个主要的类: QGraphicsScene、QGraphicsView 和 QGraphicsItem, 分别是场景、视图和图元。一个场景可以通过多个视图表现, 一个场景包括多个几何图形。

1. 场景

QGraphicsScene 类实现 Graphics View 中的场景。场景类完成如下功能:

- 提供管理大量图元的快速接口;
- 传播事件给场景中的每个图元;
- 管理图元状态, 如选择和焦点处理;
- 提供无变换的绘制功能, 如打印。

场景是 QGraphicsItem 对象的容器。通过函数 QGraphicsScene::addItem() 可以加入一个图元到场景中。图元可以通过多个函数进行检索。QGraphicsScene::items() 和一些重载函数可以返回和点、矩形、多边形或向量路径相交的所有图元。QGraphicsScene::itemAt() 返回指定点的最顶层图元。

QGraphicsScene 的事件传播体系结构将场景事件发送给图元, 同时也管理图元之间的事件传播。如果场景接收到了在某一鼠标的单击事件, 场景会把事件传给在这一点的图元。

QGraphicsScene 负责管理一些图元的状态, 如图元选择和焦点。可以通过 QGraphicsScene::setSelectionArea() 函数选择图元, 选择区域可以是任意的形状, 使用 QPainterPath 表示。要得到当前选择的图元列表可以使用函数 QGraphicsScene::selectedItems()。QGraphicsScene 还管理图元的键盘输入焦点状态。可以通过 QGraphicsScene::setFocusItem() 函数或 QGraphicsItem::setFocus() 函数来设置图元的焦点。获得当前具有焦点的图元使用函数 QGraphicsScene::focusItem()。

如果需要将场景内容绘制到特定的绘图设备, 可以使用 QGraphicsScene::render() 函数在绘图设备上绘制场景。

2. 视图

QGraphicsView 是视图窗口部件, 它使场景的内容可视化。可以连接几个视图到一个场景, 也可

以为相同的数据集提供几种不同的视口。QGraphicsView 是可滚动的窗口部件，可以提供滚动条来浏览大的场景。如果需要使用 OpenGL，可以使用 QGraphicsView::setViewport() 将视口设置为 QGLWidget。

视图接收键盘和鼠标的输入事件，并把它翻译为场景事件（将坐标转换为场景的坐标）。使用变换矩阵函数 QGraphicsView::matrix() 可以变换场景的坐标，通过这种方法可以实现场景缩放和旋转。QGraphicsView 提供 QGraphicsView::mapToScene() 和 QGraphicsView::mapFromScene() 来和场景的坐标进行转换。

3. 图元

QGraphicsItem 是图元基类。QGraphicsView 框架提供了几种标准的图元，如矩形 (QGraphicsRectItem)、椭圆 (QGraphicsEllipseItem) 和文本图元 (QGraphicsTextItem) 等。用户可以继承 QGraphicsItem 实现符合自己需要的图元。

QGraphicsItem 具有下列功能：

- 处理鼠标按下、移动、释放、双击、悬停、滚轮和右键菜单事件；
- 处理键盘输入事件；
- 处理拖放事件；
- 分组；
- 碰撞检测。

图元有自己的坐标系，也提供场景和图元，图元和图元之间的坐标变换函数。图元也可以通过 QGraphicsItem::matrix() 来进行自身的变换。图元可以包含子图元。

6.7.2 Graphics View 坐标系

Graphics View 坐标基于笛卡尔坐标系，一个图元的场景坐标具有 X 坐标和 Y 坐标。当使用没有变换的视图观察场景时，场景中的一个单元对应屏幕上的一个像素。

在 Graphics View 中有三个有效的坐标系：图元坐标、场景坐标和视图坐标。Graphics View 提供了三个坐标系之间的转换函数。在绘制图形时，Graphics View 的场景坐标对应 QPainter 的逻辑坐标，视图坐标和设备坐标相同。

1. 图元坐标

图元使用自己的本地坐标。这个坐标系通常以图元中心为原点，这也是所有变换的原点。图元坐标方向是 X 轴正方向向右，Y 轴正方向向下。创建图元后，只需要注意图元坐标就可以了，QGraphicsScene 和 QGraphicsView 会完成所有的变换。

2. 场景坐标

场景坐标是所有图元的基础坐标系。场景坐标系描述了顶层的图元，每个图元都有场景坐标和相应的包容框。场景坐标的原点在场景中心，坐标原点是 X 轴正方向向右，Y 轴正方向向下。

3. 视图坐标

视图坐标是窗口部件的坐标。视图坐标的单位是像素。QGraphicsView 视口的左上角是 (0, 0)，X 轴正方向向右，Y 轴正方向向下。所有的鼠标事件最开始都是使用视图坐标。

4. 坐标映射

在 Graphics View 框架中，经常需要将多种坐标变换，从场景到图元，从图元到图元，从视图到场



景。Graphics View 框架提供了如表 6-7 所示的多种变换函数:

表 6-7 Graphics View 框架坐标变换函数

映射函数	转换类型
<code>QGraphicsView::mapToScene()</code>	视图到场景
<code>QGraphicsView::mapFromScene()</code>	场景到视图
<code>QGraphicsItem::mapFromScene()</code>	场景到图元
<code>QGraphicsItem::mapToScene()</code>	图元到场景
<code>QGraphicsItem::mapToParent()</code>	子图元到父图元
<code>QGraphicsItem::mapFromParent()</code>	父图元到子图元
<code>QGraphicsItem::mapToItem()</code>	本图元到其他图元
<code>QGraphicsItem::mapFromItem()</code>	其他图元到本图元

6.7.3 深入 Graphics View

Graphics View 框架提供一些常见操作可直接使用,也可以进行改造以符合用户的需求。

1. 缩放和旋转

`QGraphicsView` 通过 `QGraphicsView::setMatrix()` 支持同 `QPainter` 一样的几何变换。当进行视图变换时, `QGraphicsView` 保持视图的中心。通过应用变换,可以很容易地实现缩放和旋转。

下面的例子说明如何通过缩放和旋转槽来实现对视图的缩放和旋转。

```
class View : public QGraphicsView
{
    Q_OBJECT
    ...
public slots:
    void zoomIn() { scale(1.5, 1.5); }
    void zoomOut() { scale(1 / 1.5, 1 / 1.5); }
    void rotateLeft() { rotate(-90); }
    void rotateRight() { rotate(90); }
    ...
};
```

将槽和具有 `autoRepeat` 属性的 `QToolButton` 进行连接,就可以实现连续的缩放操作。

2. 光标和工具提示

和 `QWidget` 一样, `QGraphicsItem` 支持图元特定的光标(应用 `QGraphicsItem::setCursor()`)和工具提示(应用 `QGraphicsItem::setToolTip()`)。在鼠标进入图元区域时激活相应的光标和工具提示。

3. 动画

Graphics View 支持几种不同级别的动画。可以将动画路径通过 `QGraphicsItem Animation` 和图元关联。这可以使时间线性控制的图元在所有平台上速度一致。`QGraphicsItemAnimation` 允许创建图元的路径,包括位置、旋转、缩放、扭曲、平移等操作的路径,即在不同的时候进行不同的变换。动画通常用 `QTimeLine` 来控制,也可以用 `QSlider` 来控制。

也可以创建从 `QObject` 和 `QGraphicsItem` 继承的图元, 此类图元可以设置自己的定时器, 通过 `QObject::timerEvent()` 来控制动画。

下面的代码模拟了太阳升起、落下的过程。

```
#include <QtGui>
#include <cmath>

using namespace std;
const qreal PI = 3.14159265;

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QGraphicsEllipseItem *sun = new QGraphicsEllipseItem(0, 0, 20, 20);
    sun->setBrush(Qt::red);
    sun->setPen(QPen(Qt::red));

    QTimeline *timeline = new QTimeline(10000);
    timeline->setCurveShape(QTimeline::LinearCurve);

    QGraphicsItemAnimation *animation = new QGraphicsItemAnimation;
    animation->setItem(sun);
    animation->setTimeline(timeline);

    qreal x, y;
    qreal angle = PI;
    for (int i = 0; i <= 180; ++i)
    {
        x = 200.0 * cos(angle);
        y = 200.0 * sin(angle);
        animation->setPosAt(i/180.0, QPointF(x, y));
        angle += PI/180.0;
    }

    QGraphicsScene *scene = new QGraphicsScene();
    scene->addItem(sun);

    QGraphicsView *view = new QGraphicsView(scene);
    view->resize(640, 480);
    view->show();

    timeline->start();
    return app.exec();
}
```

程序中使用了 `QTimeline` 对象控制动画。`QTimeline` 的值变化曲线设为 `QTimeline::LinearCurve`, 即 `QTimeline` 的值是线性变化的。然后分 180 步设置了每步的椭圆位置, 并开始进行 10 秒钟的动画。程序模拟了太阳沿半圆形的轨迹运动的过程。

4. OpenGL 绘制

要使用 OpenGL 绘制，可以调用 `QGraphicsView::setViewport()` 来设置 `QGLWidget` 作为 `QGraphicsView` 的视口。如果需要在 OpenGL 中打开反锯齿，可以使用 `QGLFormat::sampleBuffers()` 来使用 OpenGL 的采样缓冲区 (sample buffer)。

5. 图元组

使用图元组可以将图元组合在一起，对图元组的变换对所有子图元都有效。`QGraphicsItem` 可以处理所有子图元的事件 (使用 `GraphicsItem::setHandlesChildEvents()`)，即允许组合图元处理所有子图元的事件。

6. 实例

下面通过一个例子来说明如何使用 `Graphics View` 来绘制和管理图形。玩过“简氏舰队指挥官”的读者应该见过各种标号表示的舰艇、飞机等的作战标图。这里建立一个图形系统，能够显示类似于舰队指挥官中各种水面、水下、空中目标，分别使用不同的图形表示。每种目标具有“敌”、“我”、“不明”的属性，分别使用红、青、黄三种颜色表示。每个目标定义为一个 `QGraphicsItem`，具体定义如下：

```
#ifndef TARGET_H
#define TARGET_H

#include <QGraphicsItem>
#include <QObject>

class Target : public QObject, public QGraphicsItem
{
    Q_OBJECT

public:
    Target();

    QRectF boundingRect() const;
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
              QWidget *widget);

public:
    qreal course;
    qreal speed;
    short type;      // 空中、水面、水下
    short attribute; // 敌方、我方、不明
    QColor color;

protected:
    void contextMenuEvent(QGraphicsSceneContextMenuEvent *event);
};

#endif
```

这里定义了每个图元的航向、航速、类型、属性及显示的颜色。

由于 `QGraphicsItem` 是抽象基类，所以至少要实现两个纯虚函数 `boundingRect()` 和 `paint()`，具体实现如下：

```
#include "target.h"

#include <QGraphicsScene>
#include <QPainter>
#include <QStyleOption>

#include <math.h>

static const double Pi = 3.14159265358979323846264338327950288419717;

Target::Target()
    : speed(qrand() % 10 + 1), type(qrand() % 3), attribute(qrand() % 3)
{
    course = (qrand() % 360);
    switch(attribute) {
        case 0: // 敌方
            color.setRgb(255, 0, 0);
            break;
        case 1: // 我方
            color.setRgb(0, 255, 255);
            break;
        default: // 不明
            color.setRgb(255, 255, 0);
            break;
    }
}
```

在图元的构造函数中初始化了随机的航向、航速、类型属性，并根据敌方、我方、不明属性设置了图元的颜色。

`boundingRect()` 函数返回图元的包容框。

```
QRectF Target::boundingRect() const
{
    qreal adjust = 0.5;
    return QRectF(-20 - adjust, -22 - adjust,
                  40 + adjust, 83 + adjust);
}
```

`paint()` 函数绘制图元自身，根据不同类型的目标绘制不同的图形，代码如下：

```
void Target::paint(QPainter *painter,
                  const QStyleOptionGraphicsItem *, QWidget *)
{
    QPen pen(color);
    pen.setWidth(2);
```



```

painter->setPen(pen);
switch(type) {
case 0: //水面目标
    painter->drawEllipse(-15, -15, 30, 30);
    break;
case 1: // 水下目标
    painter->drawArc(-15, -15, 30, 30, 180*16, 180*16);
    break;
default: // 空中目标
    painter->drawLine(-15, 0, 0, -15);
    painter->drawLine(0, -15, 15, 0);
    break;
}
painter->drawLine(0, 0, int(speed*5*cos(course)),
    int(speed*5*sin(course)));
}

```

在图上可以弹出右键菜单，处理函数如下。

```

void Target::contextMenuEvent(QGraphicsSceneContextMenuEvent *event)
{
    QMenu menu;
    menu.setWindowOpacity(0.8);
    QAction *removeAction = menu.addAction(tr("武器发射"));
    QAction *markAction = menu.addAction(tr("电子干扰"));
    QAction *selectedAction = menu.exec(event->screenPos());
}

```

所有图形在一个 QGraphicsView 的视图类上显示，定义为 radarView 如下：

```

#ifndef RADAR_H
#define RADAR_H
#include <QGraphicsView>

class radarView : public QGraphicsView
{
    Q_OBJECT

public:
    radarView(QGraphicsScene * scene, QWidget * parent = 0 );
    QAction *actStrike;

public slots:
    void timerEvent(QTimerEvent *event);
};

#endif

```

视图类的实现如下：

```

#include <math.h>

```

```

#include <QAction>
#include <QMenu>
#include <QContextMenuEvent>
#include "radar.h"
#include "target.h"

radarView::radarView( QGraphicsScene * scene, QWidget * parent) :
QGraphicsView(scene, parent)
{
    startTimer(1000);
    actStrike = new QAction(tr("发射武器"),this);
}

```

为简化模型，这里假设所有的图元都是匀速直线运动，所以要定时计算图元下一次的位。

```

void radarView::timerEvent(QTimerEvent *)
{
    QList<QGraphicsItem*> itemList = items();
    QGraphicsItem *item;
    foreach(item, itemList)
    {
        Target* target = (Target*) item;
        target->setPos(target->mapToParent(target->speed *
            cos(target->course), target->speed * sin(target->course)));
    }
}

```

最后实现主程序如下。

```

#include "target.h"
#include <QtGui>
#include <QTextCodec>
#include <math.h>
#include "radar.h"

static const int TargetCount = 200;

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
    qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));

    QGraphicsScene scene;
    scene.setSceneRect(-400, -300, 800, 600);
    scene.setItemIndexMethod(QGraphicsScene::NoIndex);

    for (int i = 0; i < TargetCount; ++i) {
        Target *target = new Target;
        target->setPos(qrand() % 800 - 400,

```



```
        qrand() % 600 - 300);  
    target->setVisible(true);  
    scene.addItem(target);  
}  
  
    radarView view(&scene);  
    view.setRenderHint(QPainter::Antialiasing);  
    view.setBackgroundBrush(QColor(0,0,0));  
    view.setCacheMode(QGraphicsView::CacheBackground);  
    view.setDragMode(QGraphicsView::ScrollHandDrag);  
    view.setWindowTitle(QObject::tr("海战模拟"));  
    view.resize(800, 600);  
    view.show();  
  
    return app.exec();  
}
```

在主程序中创建了场景、视图和大批量的图元。视图的 `setBackgroundBrush()` 函数将背景设为黑色。运行效果如图 6-17 所示。



图 6-17 Graphics View 框架示例

6.8 图形图像打印

Qt 提供了跨平台的打印支持，能够使用本地打印机和远程打印机。Qt 的打印系统甚至支持直接生成 PostScript 和 PDF 文件。QPrinter 类对打印机进行了抽象，它实际上是支持打印的特殊绘图设备 (QPaintDevice)。QPrinter 支持多页和双面打印。使用 QPrinter 可以和绘制自定义的窗口部件一样完成打印操作。

6.8.1 普通打印过程

打印的过程通常是弹出打印设置对话框，用户指定打印机、纸张尺寸和其他的一些打印属性，然后进行打印。下面的代码片段实现弹出标准打印对话框。


```

QPrinter printer;
QPrintDialog *dialog = new QPrintDialog(&printer, this);
dialog->setWindowTitle(tr("打印文档"));
if (dialog->exec() != QDialog::Accepted)
    return;

```

创建了打印设备之后就是对打印设备的设置，然后就可以开始打印了。下面的代码演示了打印的过程。

```

QPainter painter;
painter.begin(&printer);
for (int page = 0; page < numberOfPages; ++page) {
    // 使用 QPainter 的绘图函数进行打印
    if (page != lastPage)
        printer.newPage();
}
painter.end();

```

程序中的 `newPage()` 函数告知打印机结束当前页打印，开始打印新的一页。

`QPrinter` 提供了函数获取纸张尺寸和可打印区域。这些数据使用逻辑坐标表示，不同于设备自身的物理坐标。可以使用 `paperRect()` 获取纸张尺寸，`pageRect()` 获取可打印区域尺寸。由于通常有页边距，所以 `pageRect()` 一般比 `paperRect()` 小。两个函数返回的区域如图 6-18 所示。

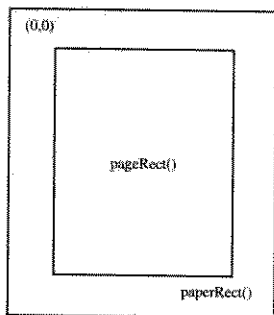


图 6-18 打印坐标系

这里给前面的基本图形绘制程序添加打印功能。

```

void MainWindow::print()
{
    QPrinter printer;
    QPrintDialog dialog(&printer, this);
    if (dialog.exec()) {
        QPainter painter(&printer);
        QRect rect = painter.viewport();
        QSize size = imageWidget->size();
        size.scale(rect.size(), Qt::KeepAspectRatio);
        painter.setViewport(rect.x(), rect.y(), size.width(),

```



```

        size.height());
    painter.setWindow(imageWidget->rect());
    painter.drawPixmap(0, 0, QPixmap::grabWidget(imageWidget,
        imageWidget->rect()));
}
}

```

程序将窗口部件上的图形用 `grabWidget()` 函数抓取到 `QPixmap` 对象中，然后将 `QPixmap` 对象绘制到打印机上。

6.8.2 特殊窗口部件的打印

一些特殊窗口部件的绘制功能是由相应的内容管理类进行管理，如 `QTextEdit` 和 `QGraphicsView` 显示的内容分别由 `QTextDocument` 和 `QGraphicsScene` 类管理。因此对这些类来说，它们的打印功能由内容管理类或特定的函数完成。这些类如表 6-8 所示。

表 6-8 不同窗口部件的打印函数

窗口部件	打印函数
<code>QGraphicsView</code>	<code>QGraphicsView::render()</code>
<code>QSvgWidget</code>	<code>QSvgRenderer::render()</code>
<code>QTextEdit</code>	<code>QTextDocument::print()</code>
<code>QTextLayout</code>	<code>QTextLayout::draw()</code>
<code>QTextLine</code>	<code>QTextLine::draw()</code>

`Graphics View` 框架通过场景的 `QGraphicsScene::render()` 函数和视图的 `QGraphicsView::render()` 函数结合就可以完成打印。这两个函数将场景和视图上的内容全部打印到任意的绘图设备。场景和视图绘制函数的差别在于一个使用场景坐标，一个使用视图坐标。`QGraphicsScene::render()` 通常用来绘制没有变换的场景，如几何数据、文本文档。`QGraphicsView::render()` 则用来实现屏幕快照，它默认的行为是将视口的数据绘制到指定的绘图设备。

当源区域和目标区域大小不相同时，源区域将会按指定的内容缩放以符合目标区域。缩放比例取决于 `Qt::AspectRatioMode`。

下面给 6.7 节的 `Graphics View` 框架的例子加上打印功能，程序如下：

```

void MainWindow::print()
{
    QPrinter printer;
    if (QPrintDialog(&printer).exec() == QDialog::Accepted) {
        QPainter painter(&printer);
        painter.setRenderHint(QPainter::Antialiasing);
        scene->render(&painter);
    }
}

```

用户可以将上面的 `scene->render(&painter)` 换为 `view->render(&painter)`，对比一下场景和视图打印的区别。

6.9 小 结

Qt 的 2D 绘图功能非常强大，它由三个核心类组成：QPainter、QPaintEngine 和 QPaintDevice。它提供了丰富的绘图函数、图像处理、绘图路径、渐变画刷、纹理画刷等多项便利的功能。Qt 的绘图操作和设备是无关的，在屏幕上和打印机上的绘图函数是一套 API。从 Qt 4.2 开始引入的 Graphics View 框架则为提供了便利的图元管理和操作功能。

第7章 拖放操作和剪贴板

拖放操作提供了友好用户的界面，通过拖放操作可以在应用程序内或应用程序间传递数据；剪贴板则提供了另外一种简便的数据交换方式。本章将介绍如何实现这两种数据交换方式。

7.1 拖放操作

拖放操作可以在应用程序间进行，也可以在应用程序内完成。Qt 为了准确地识别拖放行为，使用了两个变量设置识别拖放和单击的界限：第一个变量是 `QApplication::startDragTime`，该变量描述了用户按下鼠标多长时间才开始一个拖放操作，默认是 500 毫秒；第二个变量是 `QApplication::startDragDistance`，该变量描述了用户按下鼠标时移动多少个像素才开始拖动，默认是 4 个像素。通常情况下这些默认值是合理的，不需要改动。

7.1.1 拖放操作

为了开始一个拖动，要创建一个 `QDrag` 对象，然后调用它的 `start()` 函数。一般应该在鼠标移动一定的距离后再开始拖放操作，这样可以更准确地识别拖放操作。在窗口部件的 `mousePressEvent()` 中开始一个拖放操作，代码框架如下：

```
void MainWindow::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton)
        QDrag *drag = new QDrag(this);
        QMimeData *mimeType = new QMimeData;

        mimeType->setText(textEdit->toPlainText());
        drag->setMimeData(mimeType);
        drag->setPixmap(dragPixmap);

        Qt::DropAction dropAction = drag->start();
        .....
}
```

在程序中，使用了 `QMimeData` 对象来描述拖放的数据，`QMimeData` 是 MIME (Multipurpose Internet Mail Extension protocol) 类型数据的一个容器类。剪贴板的数据描述同样也使用 `QMimeData`。

`start()` 函数开始一个拖动，可以传递参数给 `start()`，定义初始的拖放行为是复制、移动还是其他操作。`start()` 函数完成后返回一个值说明实际的拖放行为。参数和返回值类型均为 `Qt::DropAction`，常用值的定义如表 7-1 所示。

表 7-1 拖放操作的几种类型

常 量	值	意 义
Qt::CopyAction	0x1	复制数据
Qt::MoveAction	0x2	移动数据
Qt::LinkAction	0x4	连接数据
Qt::IgnoreAction	0x0	操作被忽略

为了使目标窗口部件能够接受拖放操作, 调用 `setAcceptDrops(true)` 允许窗口部件接受拖放操作, 同时还要实现 `dragEnterEvent()` 和 `dropEvent()` 两个事件处理函数。

`dragEnterEvent()` 函数事件在拖放数据到达目标 widget 时产生, 在函数中用户只处理感兴趣的拖放数据, 如下代码实现只支持文本数据的拖放。

```
void Window::dragEnterEvent(QDragEnterEvent *event)
{
    if (event->mimeTypeData()->hasFormat("text/plain"))
        event->acceptProposedAction();
}
```

为了接收拖放数据, 还必须实现目标控件的 `dragMoveEvent()` 或 `dropEvent()` 响应函数。`dropEvent()` 函数完成解释接收的拖放数据, 并通知接收的应用做相应的处理。

下面通过将第 6 章的图像浏览器程序增加拖放操作来说明如何使用拖放操作。在这里能够通过拖动文件管理器 (如 KDE 中的 Konqueror) 中的图像文件到图像浏览器中并打开。

首先实现当拖动进入应用程序客户窗口时的事件 `dragEnterEvent(QDragEnterEvent *event)` 如下:

```
void ImageWidget::dragEnterEvent(QDragEnterEvent *event)
{
    if (event->mimeTypeData()->hasUrls()) {
        QString localFile;
        QRegExp rx("\\.(jpg|bmp|jpeg|png|xpm)$", Qt::CaseInsensitive);
        foreach(QUrl url, event->mimeTypeData()->urls()) {
            localFile = url.toLocalFile();
            if(rx.indexIn(localFile) >= 0) {
                event->accept();
                return;
            }
        }
        else {
            event->ignore();
        }
    }
    else {
        event->ignore();
    }
}
```

在这里, 通过 `QMimeData` 类的 `hasUrls()` 函数判断拖动数据是否是 URL 类型 (即 MIME 类型 `text/uri-list`)。 `QMimeData` 可以判断的一些常用 MIME 类型如表 7-2 所示。



表 7-2: MIME 类型判断函数

函 数	判断的类型
hasHtml()	text/html
hasText()	text/plain
hasUrls()	text/uri-list
hasColor()	application/x-color

由于 text/uri-list 还可以是因特网地址, 如 <http://www.broadview.com.cn>, 但这里只处理本地文件, 所以使用 `QUrl.toLocalFile()` 返回本地路径, 排除网络路径。再使用正则表达式 (在第 13 章有详细描述) 过滤扩展文件名为 jpg, bmp, jpeg, png, xpm 的文件。如果满足条件, 则用 `accept()` 函数接受拖放事件, 否则用 `ignore()` 忽略拖放事件。后者将光标变为禁止拖放的形状。

`dragMoveEvent()` 函数的内容和 `dragEvent()` 一样。为了使窗口部件能响应 drop 事件, 需要实现 `dropEvent()` 函数。

```
void ImageWidget::dropEvent(QDropEvent *event)
{
    if (event->mimeTypeData()->hasUrls()) {
        QString localFile;
        QRegExp rx{"\\.(jpg|bmp|jpeg|png|xpm)$", Qt::CaseInsensitive};
        foreach(QUrl url, event->mimeTypeData()->urls()) {
            localFile = url.toLocalFile();
            if(rx.indexIn(localFile) >= 0) {
                event->accept();
                setPixmap(localFile);
                return;
            }
        }
        event->ignore();
    }
    else {
        event->ignore();
    }
}
```

`dropEvent()` 函数和 `dragEnterEvent()` 基本一致, 只是增加了调用 `setPixmap()` 函数显示图像的语句。

7.1.2 定义新的拖放操作类型

为了实现拖放自定义的数据类型, 可以自定义 MIME 类型。所有的数据放在 `QByteArray` 的字节数组内, 然后使用自定义的数据类型, 实现代码如下所示:

```
QByteArray itemData;
QDataStream dataStream(&itemData, QIODevice::WriteOnly);
dataStream << pixmap << event->pos();

QMimeData *mimeTypeData = new QMimeData;
mimeTypeData->setData("application/mymimedata", itemData);
```

在接收放下的数据时，则使用 `QByteArray` 取出数据，然后用 `QDataStream` 读出数据。

```
QByteArray itemData = event->mimeTypeData()->data("application/mymimedata");
QDataStream dataStream(&itemData, QIODevice::ReadOnly);
```

7.1.3 Graphics View 框架下的拖放操作

在 `Graphics View` 框架中，拖放操作和 `QWidget` 中的略有不同。场景、视图和图元都可以处理自己的拖放事件。`QGraphicsView` 直接继承自 `QWidget`，所以它具备 `QWidget` 的拖放功能。当视图收到拖放操作时，将拖放事件转换为 `Graphics View` 的 `QGraphicsSceneDragDropEvent` 事件，然后将事件传递给场景。场景对这些事件进行调度，将它发送给在鼠标位置的图元。

为了开始拖放一个图元，需要创建一个 `QDrag` 对象并将指针传给这个窗口部件，然后开始拖动操作。图元可以在多个视图出现，但是只有一个视图能够进行图元的拖放操作。拖放操作通常在按下鼠标并移动的时候产生，因此可以在 `mousePressEvent()` 和 `mouseMoveEvent()` 函数中获得原始窗口部件的指针。如下例所示。

```
void MyItem::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    QMimeData *data = new QMimeData;
    data->setHtml("<h1>ItemData<h1>");

    QDrag drag(event->widget());
    drag.setMimeData(data);
    drag.start();
}
```

为了拦截场景的拖放操作，可以在 `QGraphicsItem` 的子类中重新实现 `QGraphicsScene::dragEnterEvent()` 或其他需要拦截的事件处理器。

下面通过为上一章中的海战模拟程序添加拖放功能来学习如何在 `Graphics View` 中使用拖放操作。这里要实现的是拖动我方目标到敌方目标上，表示我方对敌方的目标发动导弹攻击，同时创建一个快速的飞向敌方的空中目标（即发射的导弹）。

由于图元默认不支持拖放操作，所以要在图元的构造函数中加入 `setAcceptDrops(true)`，接下来实现图元的鼠标按下事件，开始拖动。

```
void Target::mousePressEvent(QGraphicsSceneMouseEvent * event)
{
    if((m_attribute != Us)) {
        event->ignore();
        return;
    }

    setCursor(Qt::ClosedHandCursor);
    QByteArray itemData;
    QDataStream dataStream(&itemData, QIODevice::WriteOnly);
    dataStream << mapToParent(QPointF(0, 0)); //传递场景坐标

    QMimeData *data = new QMimeData;
```



```

data->setData("application/x-dndtarget", itemData);

QDrag *drag = new QDrag(event->widget());
drag->setMimeData(data);
drag->start();
}

```

函数中首先判断是否为我方目标，如果不是则不响应拖放事件。开始拖放后，将鼠标形状设为“握住的手”（Qt::ClosedHandCursor），给用户反馈。拖放的数据是自定义的数据类型 `x-dndtarget`，内容为我方目标的场景坐标。

`dragEnterEvent()` 函数在拖放操作落在图元上时响应。

```

void Target::dragEnterEvent(QGraphicsSceneDragDropEvent * event)
{
    if(m_attribute != Foe)
        event->ignore();
    else if(event->mimeType()->hasFormat("application/x-dndtarget")) {
        event->accept();
    } else {
        event->ignore();
    }
}

```

上面的代码判断了当前图元是否属于敌方，如果不是则忽略 `dragEnterEvent` 事件。这里还对拖放数据的有效性进行了判断。

最后在放置数据时，调用 `dropEvent()` 函数。

```

void Target::dropEvent(QGraphicsSceneDragDropEvent * event)
{
    if(m_attribute != Foe)
        event->ignore();
    else if(event->mimeType()->hasFormat("application/x-dndtarget")) {
        QByteArray itemData =
            event->mimeType()->data("application/x-dndtarget");
        QDataStream dataStream(&itemData, QIODevice::ReadOnly);

        QPointF missile;
        dataStream >> missile;

        QPointF owner = mapToParent(0, 0);
        qreal dx = owner.x() - missile.x();
        qreal dy = owner.y() - missile.y();
        qreal course = atan(dy/dx);
        QPointF test = mapFromScene(missile);
        if (((test.x() > 0) && (test.y() > 0)) ||
            ((test.x() > 0) && (test.y() < 0)))
            course = Pi + course;

        Target *target = new Target(12, course, Air, Us);
    }
}

```



```

target->setPos(missile);
target->setVisible(true);
scene()->addItem(target);

event->accept();
} else {
    event->ignore();
}
}

```

函数中计算了产生的空中目标对象的角度，设置了新目标图元的属性，并将新图元加入到场景中。

7.2 使用剪贴板

使用 `QClipboard` 类可以存取窗口系统的剪贴板。`QClipboard` 支持和 `QDrag` 类相同的数据类型，使用类似的机制。获取 `QClipboard` 对象的方法如下：

```
clipboard = QApplication::clipboard();
```

在 Qt 中使用 `QMimeData` 类来表示剪贴板交换的数据。将一些常用的数据存入到剪贴板可以使用 `setText()`、`setImage()` 和 `setPixmap()` 函数。这些函数同 `QMimeData` 类中的成员类似，不同的是 `QMimeData` 类成员可以指定数据存储的位置，如果指定了存储到 `QClipboard::Clipboard` 则数据存放在剪贴板，如果指定了 `QClipboard::Selection` 则数据存放在鼠标选择的容器中（只对 X11 有效），在 Mac OS X 上，数据还可以存在 `QClipboard::FindBuffer` 中。默认情况下数据存放在剪贴板。

例如将 `QLineEdit` 中的文字复制到剪贴板可以使用：

```
clipboard->setText(lineEdit->text(), QClipboard::Clipboard);
```

上面语句中的第二个参数也可以不写，因为数据默认就是存放在剪贴板中。

不同的 MIME 类型的数据都可以使用剪贴板进行交换。通过生成一个 `QMimeData` 对象然后使用 `setData()` 函数存放数据，最后使用 `setMimeData()` 函数将数据存入剪贴板。

剪贴板底层实现机制在不同的系统上不相同，在 X11 上使用 XDND 协议，在 Windows 上使用 OLE 标准，Mac OS 上使用 Carbon 拖放管理器。XDND（Drag-and Drop Protocol for the X Window System）是 X 窗口系统广泛使用的拖放协议，它使用 MIME，所以 Qt 能够透明地支持。在 Windows 平台和 Mac 平台上，Qt 使用内部类 `QWindowsMime` 和 `QMacPasteboardMime` 类映射专有的剪贴板类和 MIME 类。

当剪贴板数据发生改变时，`QClipboard` 类发出 `dataChanged()` 信号，用户可以通过监听这个信号获取剪贴板的变化信息。

在 X11 平台上，Qt 也支持 Motif 风格的拖放协议。Motif 的拖放协议仅允许用户在 `QDropEvent` 中接收请求的数据，在 `QDragMoveEvent` 中不能获得拖放数据。

下面通过给第 6 章的图像浏览器增加复制图像到剪贴板的功能来演示如何使用剪贴板。首先增加一个 `QAction` 处理复制事件。

```

copyAct = new QAction(QIcon(":/images/copy.png"), tr("Copy"), this);
copyAct->setShortcut(QKeySequence::Copy);
connect(copyAct, SIGNAL(triggered()), this, SLOT(copy()));

```

然后将该 `QAction` 加入到菜单和工具栏，`copyAct` 的响应槽函数 `copy()` 实现如下：



```
void MainWindow::copy()
{
    QPixmap pix = imageWidget->getPixmap();
    clipboard->setImage(pix.toImage());
}
```

可以看到使用剪贴板相当简单。

7.3 小 结

Qt 支持基于 `QMimeData` 类的拖放操作和剪贴板操作，可以定义自己的 `QMimeData` 数据类型传递数据。拖放操作既能在应用程序内部完成，也能在应用程序之间完成。对于 `Graphics View` 和 `InterView` 框架，拖放操作在实现上有一些特殊的方式以实现特定的需求。

第 8 章 文件处理

Qt 提供了 `QFile` 类来进行文件操作。为了更方便地处理文本文件和二进制文件，Qt 还提供了 `QTextStream` 类和 `QDataStream` 类。处理临时文件可以使用 `QTemporaryFile`，获取文件信息可以使用 `QFileInfo`，处理目录使用 `QDir`。监视文件和目录变化则可以使用 `QFileSystemWatcher`。

8.1 读写文本文件

`QFile` 类提供了读写文件的接口。`QFile` 类可以读写文本文件、二进制文件和 Qt 的资源文件。也可以使用更方便的 `QTextStream` 和 `QDataStream` 类读取文本文件和二进制文件。

要打开一个文件，可以在构造函数中指定文件名，也可以在任何时候使用 `setFileName()` 函数设置文件名。打开文件使用 `open()` 函数，关闭文件使用 `close()` 函数。

在 `QFile` 中可以使用从 `QIODevice` 中继承的 `readLine()` 函数读取文本文件的一行。如：

```
QFile file("zeki.txt");
if (file.open(QIODevice::ReadOnly)) {
    char buffer[2048];
    qint64 lineLen = file.readLine(buffer, sizeof(buffer));
    if (lineLen != -1) { // 读取成功
        qDebug() << buffer;
    }
}
```

上面代码中 `open()` 函数以只读方式打开文件，只读方式用参数 `QIODevice::ReadOnly`，只写方式参数为 `QIODevice::WriteOnly`，读写参数是 `QIODevice::ReadWrite`。

如果读取成功，`readLine` 函数返回实际读取的字节数；如果读取失败则返回“-1”。

`QTextStream` 提供了更为方便的接口来读写文本。`QTextStream` 可以操作 `QIODevice`、`QByteArray` 和 `QString`。使用 `QTextStream` 的流操作符，可以方便地读写单词、行和数字。为了产生文本，`QTextStream` 提供了填充、对齐和数字格式化的格式选项。下面的代码演示了如何使用格式化选项。

```
QFile data("test.txt");
if (data.open(QFile::WriteOnly | QFile::Truncate)) {
    QTextStream out(&data);
    out << QObject::tr("成绩: ") << qSetFieldWidth(10) << left << 90 << endl;
}
```

`open()` 函数中的参数 `QFile::Truncate` 表示将原来文件中的内容清空。在输出时将格式设为左对齐，占 10 个字符位置。

代码中的 `qSetFieldWidth()` 是设置字段宽度的格式化函数。`QTextStream` 还可以使用其他的一些格式化函数，如表 8-1 所示。



表 8-1 QTextStream 的格式化函数

函 数	功 能
qSetFieldWidth(int Width)	设置字段宽度
qSetPadChar(QChar ch)	设置填充字符
qSetRealNumberPrecision(int precision)	设置实数精度

QTextStream 定义了类似于<iostream>中的一些流操作符 (manipulator)，如上面代码中的 left 操作符。表 8-2 给出了这些流操作符。

表 8-2 QTextStream 的流操作符

操 作 符	作 用
bin	设置读写的整数为二进制数
oct	设置读写的整数为八进制数
dec	设置读写的整数为十进制数
hex	设置读写的整数为十六进制数
showbase	强制显示进制前缀，如十六进制 (0x)，八进制 (0)，二进制 (0b)
forcesign	强制显示符号 (+, -)
forcepoint	强制显示小数点
noshowbase	不显示进制前缀
noforcesign	不显示符号
uppercasebase	显示大写的进制前缀
lowercasebase	显示小写的进制前缀
uppercasedigits	用大写字母表示
lowercasedigits	用小写字母表示
fixed	固定小数点表示
scientific	科学计数法表示
left	左对齐
right	右对齐
center	居中
endl	换行
flush	清除缓冲

用户使用这些格式化函数和流操作符能够设置需要的输出格式。

在 QTextStream 中使用的默认编码是 QTextCodec::codecForLocale() 函数返回的编码，同时能够自动检测 Unicode。也可以使用 QTextStream::setCodec(QTextCodec *codec) 函数设置流的编码，在第 20 章将会讲到这种用法。

8.2 操作二进制文件

QDataStream 类提供了将二进制文件串行化的功能。QDataStream 实现了 C++ 的基本数据类型的串行化，如 char, short, int, char* 等。更复杂的数据类型串行化通过将数据类型分解为基本的数据类型来完成。

下例用来写二进制数据到数据流。

```
QFile file("binary.dat");
file.open(QIODevice::WriteOnly);
```

```

QDataStream out(&file);           //将数据序列化
out << QObject::tr("坐标: ");    // 字符串序列化
out << (qint32)42 << (qint32)96; // 整数序列化

```

将上面写入文件的数据读入的过程是:

```

QFile file("binary.dat");
file.open(QIODevice::ReadOnly);
QDataStream in(&file);           // 从文件中读出数据
QString str;
qint32 x, y;
in >> str >> x >> y;           // 获取字符串和整数

```

每一个条目都以定义的二进制格式写入文件。Qt 中的很多类型, 包括 QBrush, QColor, QDateTime, QFont, QPixmap, QString, QVariant 等都可以写入数据流。

从 Qt 1.0 开始, 就有相应版本的 QDataStream, 而且随着 Qt 的版本升级继续更新。读写复合类型时保证版本相同非常重要。可以通过 version() 函数来获得版本号。

Qt 版本和 QDataStream 版本对照如表 8-3 所示。

表 8-3 QDataStream 的版本定义

Qt 版本	常量	QDataStream 版本
Qt 4.3	QDataStream::Qt_4_3	9
Qt 4.2	QDataStream::Qt_4_2	8
Qt 4.0, 4.1	QDataStream::Qt_4_0	7
Qt 3.3	QDataStream::Qt_3_3	6
Qt 3.1, 3.2	QDataStream::Qt_3_1	5
Qt 3.0	QDataStream::Qt_3_0	4
Qt 2.1, 2.2, 2.3	QDataStream::Qt_2_1	3
Qt 2.0	QDataStream::Qt_2_0	2
Qt 1.x	QDataStream::Qt_1_0	1

可以使用 setVersion() 来设定文件格式的版本, 以下代码将文件的版本设为 4.3 版。

```

QDataStream out(file);
out.setVersion(QDataStream::Qt_4_3);

```

如果需要读取原始数据可以使用 readRawdata() 读取数据到预先定义好的 char * 缓冲区, 写原始数据使用 writeRawData()。读写原始数据需要对数据进行编码和解码。

下面的例子演示了使用 QDataStream 进行读写文件的过程。

```

#include <iostream>
#include <QtCore>

using namespace std;

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    QFile file("binary.file");

```

```

file.open(QIODevice::WriteOnly | QIODevice::Truncate);
QDataStream out(&file);
out << QString("caizhiming");
out << QDate::fromString("1976/01/03", "yyyy/MM/dd");
out << (qint32)21;
file.close();

file.setFileName("binary.file");
if(!file.open(QIODevice::ReadOnly)) {
    cout << "打开文件错误!";
    return 1;
}
QDataStream in(&file);
QString name;
QDate birthday;
qint32 age;
in >> name >> birthday >> age;
QDebug() << name << birthday << age;
cout << qPrintable(name) << '\t' <<
    qPrintable(birthday.toString("yyyy MMM dd dddd"))
    << "\t" << age << endl;
file.close();
return 0;
}

```

在例子中, `QDataStream` 类写入了姓名 (`QString`)、生日 (`QDate`) 和年龄 (`qint32`) 三个数据, 读取时使用相同的类型读出。

`QDataStream` 可以读取任意的以 `QIODevice` 为基类的类生成的对象产生的数据, 如 `QTcpSocket`, `QUdpSocket`, `QBuffer`, `QFile`, `QProcess` 等类的的数据。可以使用 `QDataStream` 在 `QAbstractSocket` 一端写数据, 另外一端使用 `QDataStream` 读取数据, 这样就免去了烦琐的高低字节转换工作。

8.3 临时文件

实际应用中有时可能需要使用临时文件。Qt 中提供了 `QTemporaryFile` 类来操作临时文件。`QTemporaryFile` 可以安全地创建一个独一无二的临时文件。临时文件使用 `open()` 来创建, Qt 可以保证临时文件名不会重复。在临时文件对象销毁后, 将自动删除该临时文件。

临时文件通过 `close()` 关闭后还可以再打开。只要临时文件对象还没有销毁, 临时文件就一直存在并由 `QTemporaryFile` 内部保持打开。

系统的临时目录可以通过 `QDir::tempPath()` 来获取。在 UNIX/Linux 系统上临时目录通常是 `/tmp`, 在 Windows 上则是由环境变量 `TEMP` 或 `TMP` 指定。

8.4 目录操作和文件管理

8.4.1 目录操作

`QDir` 类具有存取目录结构和内容的能力。使用 `QDir` 可以操作目录、存取目录或文件信息、操作

底层文件系统，而且还可以存取 Qt 的资源文件。

Qt 使用 “/” 作为通用的目录分隔符和 URL 路径分隔符。如果在程序中使用 “/” 作为目录分隔符，Qt 会其自动转换为符合底层操作系统的分隔符（如 Linux 使用 “/”，Windows 使用 “\”）。

QDir 可以使用相对路径或绝对路径指向一个文件。isRelative() 和 isAbsolute() 函数可以判断 QDir 对象使用的是相对路径还是绝对路径。如需要将一个相对路径转换为绝对路径，使用 makeAbsolute() 函数。

目录的路径可以通过 path() 函数返回，通过 setPath() 函数设置新路径，绝对路径使用 absolutePath() 返回。目录名可以使用 dirName() 获得，它通常返回绝对路径中的最后一个元素。如果 QDir 指向当前目录，则返回 “.”。目录的路径可以通过 cd() 和 cdUp() 改变。可以使用 mkdir() 创建目录，rename() 改变目录名。

判断目录是否存在可以使用 exists()，目录的属性可以使用 isReadable()、isAbsolute()、isRelative() 和 isRoot() 来获取。目录下有很多条目，包括文件、目录和符号链接，总的条目数可以使用 count() 来统计。entryList() 可以返回目录下所有条目组成的字符串链表。文件可以使用 remove() 函数删除，删除目录用 rmdir()。

下面通过实现类似于 Linux 的 du 命令来展示目录操作的使用。Linux 的 du 可以获得目录及其子目录占用的磁盘空间大小。本例是获取目录及子目录中所有文件的大小。主程序如下：

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    QStringList args = app.arguments();
    QString path;
    if (args.count() > 1)
        path = args[1];
    else
        path = QDir::currentPath();
    qDebug() << path;
    du(path);
    return 0;
}
```

主程序接受命令行参数作为统计的目录，如不提供则使用当前目录。函数 du() 完成统计目录及子目录中所有文件的大小。函数实现如下：

```
qint64 du(const QString &path)
{
    QDir dir(path);
    qint64 size = 0;
    foreach (QFileInfo fileInfo, dir.entryInfoList(QDir::Files))
        size += fileInfo.size();
    foreach (QString subDir,
        dir.entryList(QDir::Dirs | QDir::NoDotAndDotDot))
        size += du(path + QDir::separator() + subDir);
    char unit = 'B';
    quint64 curSize = size;
    if (curSize > 1024) {
```



```

        curSize /= 1024;
        unit = 'K';
        if(curSize > 1024) {
            curSize /= 1024;
            unit = 'M';
            if(curSize > 1024) {
                curSize /= 1024;
                unit = 'G';
            }
        }
    }

    cout << curSize << unit << "\t" << QPrintable(path) << endl;
    return size;
}

```

在函数 `du()` 中, `entryInfoList(QDir::Files)` 函数返回文件信息, 然后根据这些信息计算文件大小。接下来判断是否有子目录, 如果有, 则递归计算。 `dir.entryList(QDir::Dirs | QDir::NoDotAndDotDot)` 返回所有子目录并过滤掉 “.” 和 “..” 目录。因为在 Windows 和 Linux 平台上, 目录分隔符不一样, Windows 使用的是 “\”, Linux 使用的是 “/”。为了解决这个差异, 使用了 `QDir::separator()` 函数来返回特定平台的目录分隔符。

Qt 4.3 中新引入 `QDirIterator` 类也可以完成枚举目录的功能, 读者可以使用 `QDirIterator` 类改写上面的程序。

在 Qt 4.3 中还引入了文件搜索前缀的概念, 文件搜索前缀是由至少两个字符组成(不能和 Windows 的驱动盘符混淆), 用来搜索指定文件的路径。下面的代码指定了文档的搜索路径。

```

QDir::setSearchPaths("docs", QStringList("C:\\My Documents"));
QDir::addSearchPaths("docs", QStringList("D:\\Documents"));
QFile file("docs:qt4.doc");

```

在这个示例代码中, 首先定义了 “docs” 的搜索前缀为 “C:\\My Documents”, 然后又加入了 “D:\\Documents” 目录, 这样打开 `qt4.doc` 文件时会在这两个目录中查找。

8.4.2 文件管理

`QFileInfo` 类提供与系统无关的文件信息。它能提供文件名和路径, 存取权限, 以及文件是否为目录或符号链接, 文件大小以及最后的修改/读取时间等。 `QFileInfo` 也能从 Qt 的资源中获取信息。

`QFileInfo` 可以使用相对路径或绝对路径。文件名可以在 `QFileInfo` 的构造函数中传递, 也可以使用 `setFile()` 函数指定。要判断一个文件是否存在, 使用 `exists()` 函数, 文件大小可以通过 `size()` 函数获取。文件类型可以通过 `isFile()`, `isDir()` 和 `isSymLink()` 来获取。 `symLinkTarget()` 函数返回符号链接所指向的真正文件名。

一些 `QFileInfo` 函数要查询文件系统, 但基于性能原因部分函数仅操作文件名本身, 如返回一个相对路径的文件名函数。为了提高性能, `QFileInfo` 将文件的一些信息进行了缓存。由于文件有可能被其他的用户或程序, 甚至同一个程序的其他部分改变, `QFileInfo` 提供 `refresh()` 函数来刷新文件信息。如果需要 `QFileInfo` 每次从文件系统读取信息, 而不是从缓存读取, 可以使用 `setCaching(false)` 关闭缓存。

在 UNIX (包括 Mac OS X) 上, 符号链接和符号链接所指向的文件返回的 `size()` 值是相同的。因为 UNIX 处理链接是透明的, 所以使用 `QFile` 打开符号链接实际上是打开了符号链接所指向的目标。在 Windows 上, 符号链接 (快捷方式) 是 `.lnk` 文件, 使用 `size()` 返回的大小是快捷方式自身的大小。

使用 QFile 打开快捷方式也是打开快捷方式的.lnk 文件。

文件名和目录可以通过 path()和 fileName()分解。fileName()返回的部分可以通过 baseName()和 extension()来获得主文件名和扩展文件名。文件的操作日期可以通过 created()、lastModified()和 lastRead()获取。文件的存取权限可以通过 isReadable()、isWritable()和 isExecutable()获取。文件的属主可以通过 owner()、ownerId()、group()和 groupId()获取。文件的权限和属主也可以通过 permission()一起读取。

8.5 监视文件系统变化

在 Qt 中可以使用 QFileSystemWatcher 类来监视文件和目录的改变。使用 addPath()函数来监视指定的文件和目录, 如果需要监视多个目录, 可以使用 addPaths()来加入监视。要移除不需要监视的目录, 可以使用 removePath()和 removePaths()函数。

当监视的文件被修改或删除时, 产生一个 fileChanged()信号。如果被监视的目录被改变或删除, 产生 directoryChanged()信号。下例实现了监视指定目录的功能, 主程序如下:

```
#include <iostream>
#include <QtCore>
#include <QtGui>
#include "Msg.h"

using namespace std;

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

    Msg msg;
    msg.show();

    return app.exec();
}
```

在主程序中使用了类 Msg 来实现监视目录, Msg 类的构造函数如下:

```
Msg::Msg()
{
    QFont font;
    font.setPointSize(24);
    setFont(font);
    QStringList args = QApplication::arguments();
    QString path;
    if (args.count() > 1)
        path = args[1];
    else
        path = QDir::currentPath();
    label = new QLabel();
    label->setText(tr("监视的目录: ") + path);
}
```



```

QVBoxLayout *layout = new QVBoxLayout;
layout->addWidget(label);
setLayout(layout);

fsWatcher.addPath(path);
connect(&fsWatcher, SIGNAL(directoryChanged(QString)),
        this, SLOT(directoryChanged(QString)));
}

```

在构造函数中读取命令行指定的目录作为监视目录，如果没有指定则监视当前目录。使用 connect() 函数将目录的 directoryChanged 信号和响应函数连接。

响应函数简单地使用消息框提示用户目录发生了变化，代码如下：

```

void Msg::directoryChanged(QString path)
{
    QMessageBox::information(NULL, tr("目录发生变化"), path);
}

```

8.6 文件引擎

Qt 的 QDir、QFile 和 QFileInfo 类在内部都使用一个类——QAbstractFileEngine。通过继承 QAbstractFileEngine 类，可以编写自己的文件处理函数，然后继承 QAbstractFileEngineHandler 类注册自己的文件引擎，这样就可以使用自己的文件读取引擎了。

QAbstractFileEngineHandler 是创建 QAbstractFileEngine 的类工厂。当打开一个文件时，Qt 通过内部注册的文件引擎链表，选择合适的文件引擎并创建相应的引擎对象。

为了安装一个特定的文件引擎，必须继承 QAbstractFileEngineHandler 并实现 create() 函数。实例化引擎时 Qt 自动注册该引擎，最后注册的引擎比之前注册的优先级高。

如果想实现一个读取 tar 文件的引擎，可以从 QAbstractFileEngineHandler 类继承。

```

class TarEngineHandler : public QAbstractFileEngineHandler
{
public:
    QAbstractFileEngine *create(const QString &fileName) const;
};

```

create() 函数返回 TarEngine 对象，TarEngine 是文件引擎，是真正的文件处理类。

```

QAbstractFileEngine *TarEngineHandler::create(const QString &fileName) const
{
    return fileName.toLower().endsWith(".tar") ? new TarEngine(fileName) : 0;
}

```

8.7 小结

Qt 提供了通用的文件处理类 QFile 以及处理文本的 QTextStream 和处理二进制数据的 QDataStream 类，这些流操作极大地方便了对文件的读取存储。对文件信息和目录进行操作的类是 QFileInfo、QDir 和 QDirIterator。要监视文件和目录变化，则可以使用 QFileSystemWatcher 类。最后，可以通过继承 QAbstractFileEngineHandler 类来创建自己的文件处理引擎。

第9章 网络

网络编程是应用程序开发的重要内容，与 OSI（Open System Interconnect Reference Model，开放式系统互联参考模型）七层参考模型^①略有不同的是，实际编写网络应用程序时使用的是应用层、传输层、网络层和链路层四层模型，如图 9-1 所示。Linux 操作系统虽然提供了一层统一的套接字抽象用于编写不同层次的网络程序，但使用套接字进行网络编程比较烦琐，有时甚至需要引用底层操作系统的相关数据结构，初学者不易上手。Qt 为此提供了一个全新的网络模块——QtNetwork，它大大降低了网络程序开发的难度，从而使得那些对网络编程不甚了解的初学者也可以很容易地编写网络程序。

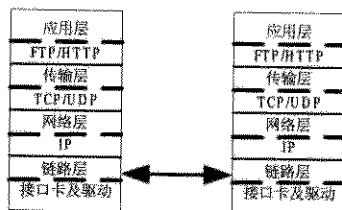


图 9-1 网络编程模型

9.1 FTP 客户端

文件传输协议 FTP（File Transfer Protocol）用于从一台主机到另一台主机传送文件，它采用两个 TCP 连接来传输一个文件，如图 9-2 所示。当用户启动与远程主机间的一个 FTP 会话时，FTP 客户端首先发起建立一个与 FTP 服务器端口号 21 之间的 TCP 控制连接，然后通过该控制连接把用户名和口令发送给服务器。用户执行的一些命令也由客户端通过控制连接发送给服务器，例如改变远程目录的命令等。当用户请求传送文件时，FTP 将在服务器端口号 20 上打开一个 TCP 数据连接，用于文件传输。需要注意的是，控制连接在整个用户会话期间一直打开，而数据连接则有可能为每次文件传送请求重新打开一次。

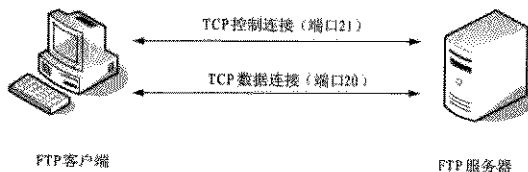


图 9-2 FTP 会话模型

^① OSI 七层参考模型是：应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。

Qt 提供了一个 QFtp 类用于构建 FTP 客户端程序, 用户使用该类可以很容易地实现绝大多数 FTP 操作。为了保证用户界面在执行 FTP 操作时的持续响应, QFtp 类以异步方式工作, 也就是说, 当执行诸如 get() 或 put() 等函数调用时, 这些调用会立即返回, 真正的数据传输直到程序控制权返回 Qt 事件循环后才会发生, 这一点十分重要。

下面来看一个简单的基于控制台的 FTP 客户端实例 FtpLogin, 这个例子演示了 FTP 的登录过程, 从中可以看到 QFtp 类的基本工作流程。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr( QTextCodec::codecForName("gb18030"));
    QStringList args = app.arguments();
    if (args.count() != 2){
        qDebug() << QObject::tr("用法: ftplogin url") << endl;
        << QObject::tr("例:") << endl;
        << QObject::tr(" ftplogin ftp://ftp.trolltech.com");
        return -1;
    }

    FtpLogin ftpClient;
    if (!ftpClient.logIn(QUrl(args[1]))){
        return -1;
    }
    QObject::connect(&ftpClient, SIGNAL(done()), &app, SLOT(quit()));
    return app.exec();
}
```

由于 FtpLogin 是一个控制台程序, 主函数 (main 函数) 中使用了 QApplication 类而不是常用的 QApplication 子类, 这样做可以避免链接庞大的 QtGui 库。当用户在控制台输入如: “FtpLogin ftp://ftp.example.com” 时, app.arguments() 函数返回命令参数并将其存储于 QStringList 型对象 args 中, 其中参数 0 为程序名, 参数 1 为程序所带的第一个实参, 即 args[0] 的值为 FtpLogin, args[1] 的值为 ftp://ftp.example.com。main() 函数的主要功能是构造一个 FtpLogin 对象, 该对象负责以传入的 args[1] 参数作为 FTP 站点地址进行登录, 如果一切调用正常, 主函数就进入事件循环, 并依次处理产生的 QFtp 事件, 直到本次 FTP 会话结束。

FtpLogin 类完成了所有实际的 FTP 操作, 定义如下:

```
class FtpLogin : public QObject
{
    Q_OBJECT
public:
    FtpLogin(QObject *parent = 0);
    ~FtpLogin();
    bool logIn(const QUrl &);
signals:
    void done();
private slots:
    void ftpDone(bool);
    void ftpCommandStarted(int);
};
```




```

connectId = ftp.connectToHost(url.host(),url.port(21));
loginId = ftp.login();
closeId = ftp.close();
return true;
}

```

登录函数 `login(const QUrl)` 首先检查输入的 `url` 地址是否合法, 协议是否正确, 如果一切正常则顺序执行连接、登录和关闭操作。 `connectToHost(url.host(),url.port(21))` 以输入的 `url` 主机和端口 21 为参数连接服务器; `login()` 不带任何参数, 表示以匿名方式登录; `close()` 则用于结束本次会话。三个操作分别返回本操作的唯一 ID, 并将其记录在 `connectId`、`loginId` 和 `closeId` 成员变量中, 用于异步方式下追踪这三个操作的执行情况。

```

void FtpLogin::ftpcommandStarted(int id)
{
    if(id == connectId){
        qDebug()<<QObject::tr("连接中...")<<endl;
    }else if(id == loginId){
        qDebug()<<QObject::tr("登录中...")<<endl;
    }else if(id == closeId){
        qDebug()<<QObject::tr("关闭中...")<<endl;
    }
}

```

每当执行新操作时, QFtp 通过 `commandStarted(int)` 信号执行 `ftpcommandStarted(int)` 函数, 参数表示本次操作的 ID。接下来用这个 ID 值与 `login(const QUrl &url)` 函数记录的各种操作 ID 值比较, 并显示其执行状况。

```

void FtpLogin::ftpcommandFinished(int id,bool error)
{
    if(error){
        qDebug()<<QObject::tr("操作错误, 程序退出! ");
        return;
    }

    if(id == connectId){
        qDebug()<<QObject::tr("已连接")<<endl;
    }else if(id == loginId){
        qDebug()<<QObject::tr("已登录")<<endl;
    }else if(id == closeId){
        qDebug()<<QObject::tr("已关闭")<<endl;
    }
}

```

QFtp 在每次 FTP 操作结束后发送 `commandFinished(int,bool)` 信号, 并调用 `ftpcommandFinished(int id,bool error)` 函数执行。它与上一个函数 `ftpcommandStarted(int id)` 类似, 只是多了一个表示执行结果的 `bool` 型参数。

```

void FtpLogin::ftpDone(bool error)
{

```

```

if(error){
    qDebug()<<tr("错误:")<<qPrintable(ftp.errorString())<<endl;
}
emit done();
}

```

最后一个槽函数 `ftpDone(bool error)` 十分简单，它在所有 FTP 操作结束后调用。它将发出 `done()` 信号用于结束整个应用程序，如果有错误还将打印错误信息。

为了链接 `QtNetwork` 库，应用程序还必须在 `.pro` 文件中加入如下一行：

```
QT += network
```

通过上面的例子可以发现使用 `QFtp` 构建 FTP 客户端其实十分简单，只需要选择其发出的有特定含义的信号进行处理就可以了，本章的所有程序都是这样做的，有区别的只是使用了不同的协议类而已。下面看一个比较复杂的例子，它实现了一个有界面的 FTP 客户端，可以用来登录、浏览站点，并下载文件，客户端界面如图 9-4 所示。

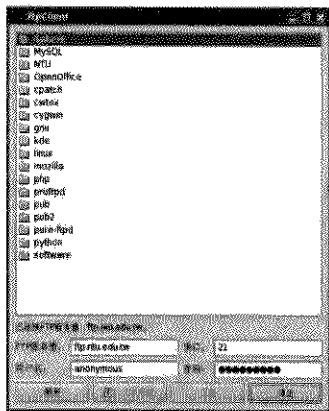


图 9-4 FTP 客户端

为了突出重点，`FtpClient` 示例的界面采用 Qt 设计器绘制，类 `FtpClientWindow` 采用多重继承的方式从 `QDialog` 类和绘制的 `Ui::FtpClientWindowClass` 派生，用来处理用户输入和界面显示，这部分内容本书在前面已经介绍过，这里不再赘述，详见示例代码。所有的 FTP 操作逻辑均由 `FtpClient` 类提供。

```

class FtpClient : public QObject
{
    Q_OBJECT

public:

    FtpClient(QObject *parent);
    ~FtpClient();

    bool isDir(QString& fileName){return isDirectory.value(fileName);};
}

```

```

QString getStatus(){return ftpStatus;};
void ftpConnect();
void changeDir(const QString&);
void getFile(const QString&);
signals:
.....
public slots:
    void ftpCommandFinished(int commandId, bool error);
    void updateDataTransferProgress(qint64 readBytes,
        qint64 totalBytes);
    void addToList(const QUrlInfo& );
    void cancelDownload();
private:

    QHash<QString, bool> isDirectory;
    QString currentPath;
    QString downloadFileName;
.....
    QFtp *ftp;
    QFile *file;
    QProgressDialog *progressDialog;
};

```

与前一个示例相同，FtpClient 也从 QObject 继承，并使用 Q_OBJECT 宏。私有成员变量 isDirectory 以哈希表的形式记录 FTP 站点当前路径下的所有项是文件还是目录；ftp 变量执行所有 QFtp 操作；file 变量用于下载文件；currentPath 变量记录当前 FTP 路径；downloadFileName 变量记录下载文件名；而 progressDialog 用于显示下载文件的进度。略去的部分为通知界面响应的相关消息和从界面获取服务器地址、端口、用户、密码，以及状态显示的相关函数和变量，详见本书代码。

```

FtpClient::FtpClient(QObject *parent)
: QObject(parent), ftp(NULL), file(NULL)
{
    progressDialog = new QProgressDialog((QWidget *)parent);
}

```

构造函数十分简单，仅仅创建一个用于显示下载进度的进度条并为 ftp 和 file 变量赋初值。

```

void FtpClient::ftpConnect()
{
    if (ftp == NULL) {
        .....
        ftp = new QFtp(this);
        connect(progressDialog, SIGNAL(canceled()),
            this, SLOT(cancelDownload()));
        connect(ftp, SIGNAL(commandFinished(int, bool)),
            this, SLOT(ftpCommandFinished(int, bool)));
        connect(ftp, SIGNAL(listInfo(const QUrlInfo &)),
            this, SLOT(addToList(const QUrlInfo &)));
        connect(ftp, SIGNAL(dataTransferProgress(qint64, qint64)),

```



```

        this, SLOT(updateDataTransferProgress(qint64, qint64)));
ftp->connectToHost(serverName,serverPort.toUShort());
ftp->login(userName,password);
ftp->list();

.....
}
else {
    ftp->abort();
    ftp->deleteLater();
    ftp = NULL;

    .....
}
}
}

```

ftpConnect()函数完成建立或终止FTP会话操作,并记录状态,发送 cmdConnected()信号通知界面显示相应状态。如果是建立FTP会话(ftp变量为空时),需要关联四个信号和槽,其中 commandFinished(int, bool)信号已经在上一例介绍过了; listInfo(const QUrlInfo &)信号在 QFtp 类发现新的FTP目录项时发出,通常由 list()操作引发,其 QUrlInfo 型参数存储了URL的相关信息,包括是否可读、可写,是否是目录、文件、符号链接等; dataTransferProgress(qint64, qint64)信号用于指示上传或下载的进度,在其槽函数中直接用进度条显示出来; canceled()信号在进度条上的取消按钮按下后产生,在这里只是简单的调用 QFtp 类的 abort()函数终止操作。中断FTP会话相对简单,首先调用 abort()终止操作,再调用 deleteLater()函数,该函数会在返回Qt事件循环后删除 QFtp 对象。

```

void FtpClient::addToList(const QUrlInfo &urlInfo)
{
    isDirectory[urlInfo.name()] = urlInfo.isDir();

    .....
}

```

每当 QFtp 发现新的目录项时, addToList(const QUrlInfo &urlInfo)函数将被调用,通过其携带的 QUrlInfo 型参数的 isDir()函数判断该项是否目录还是文件,并以名/值对的形式存储在 QMap<QString, bool>型私有变量 isDirectory 中。

```

void FtpClient::changeDir(const QString& dir)
{
    if(dir.isEmpty())//返回父目录
    {
        .....
        isDirectory.clear();
        currentPath = currentPath.left(currentPath.lastIndexOf('/'));
        if (currentPath.isEmpty()) {
            ftp->cd("/");

            .....
        } else {
            ftp->cd(currentPath);
        }
        ftp->list();
    }
}

```

```

else//进入子目录
{
    if (isDirectory.value(dir)) {
        isDirectory.clear();
        currentPath += "/" + dir;
        ftp->cd(dir);
        ftp->list();
        .....
    }
}
}

```

函数 `changeDir(const QString&)` 用来改变 FTP 站点的当前路径，具体路径名由其参数指明。如果参数为空，表示返回上级目录，此时首先必须清空当前记录的所有目录项名/值对，然后将当前路径 `currentPath` 指向其父目录，并通过 `QFtp` 的 `cd()` 函数完成路径改变操作。需要注意的是，如果修改后的当前路径为空，则表明已到达 FTP 站点根目录。最后通过 `QFtp` 的 `list()` 函数列出路径改变后的所有目录项，同时它触发的槽函数 `addToList(const QUrlInfo&)` 还会生成新的目录项名/值对。如果参数非空，表示进入子目录，首先需在记录的当前目录项名/值对中查找传入的参数是否为一个子路径，如果是，先清空当前目录项名/值对，并在当前路径 `currentPath` 上追加该子路径，最后执行 `cd()` 和 `list()` 操作进入子目录并生成新的目录项名/值对。

```

void FtpClient::getFile(const QString& fileName)
{
    if (QFile::exists(fileName)) {
        QMessageBox::information(0, tr("FTP"),
            tr("%1 文件已存在。"),
            .arg(fileName));
        return;
    }
    file = new QFile(fileName);
    if (!file->open(QIODevice::WriteOnly)) {
        QMessageBox::information(0, tr("FTP"),
            tr("无法保存文件%1: %2."),
            .arg(fileName).arg(file->errorString()));
        delete file;
        return;
    }
    ftp->get(fileName, file);
    progressDialog->setLabelText(tr("下载文件%1...").arg(fileName));
    progressDialog->show();
    .....
}

```

函数 `getFile(const QString&)` 用于将 FTP 站点上的文件下载到本地，参数指明下载的文件名。如果该文件在本地的当前路径下不存在，就在当前路径下新建一个同名文件，并以只写方式打开，否则报错返回。`QFtp` 的 `get(fileName, file)` 函数完成文件下载，并通过进度条 `progressDialog` 显示进度。

```
void FtpClient::updateDataTransferProgress(qint64 readBytes,
                                           qint64 totalBytes)
{
    progressDialog->setMaximum(totalBytes);
    progressDialog->setValue(readBytes);
}

```

槽函数 `updateDataTransferProgress(qint64, qint64)` 会在 `QFtp` 类执行文件下载操作时通过 `dataTransferProgress(qint64, qint64)` 信号触发调用。其第一个参数表示已下载的大小，第二个参数表示总文件大小，进度条指示通过这两个参数更新。

```
void FtpClient::ftpCommandFinished(int /*commandId*/, bool error)
{
    if (ftp->currentCommand() == QFtp::Login) {
        .....
    }
    if (ftp->currentCommand() == QFtp::Get) {
        .....
    } else if (ftp->currentCommand() == QFtp::List) {
        .....
    }
}

```

函数 `ftpCommandFinished(int commandId, bool error)` 在上例中已经介绍过，所不同的只是本例中没有追踪每一个操作的执行情况。取而代之的是通过 `QFtp` 类的 `currentCommand()` 函数追踪同一类操作的执行，这在有大量 FTP 操作的情况下是一种很好的选择。追踪 FTP 操作的另一种方法是使用 `stateChanged()` 信号，该信号在连接状态改变时发出。这些状态包括 `QFtp::ConnectToHost`、`QFtp::Get`、`QFtp::List` 等。函数体省去的部分主要用来修改显示状态和刷新界面，这里不再赘述。至此，一个较完整的 FTP 客户端已经完成。

`QFtp` 类还提供了许多其他常用 FTP 操作，包括 `put()`、`remove()`、`mkdir()`、`rmdir()` 和 `rename()` 等，这些函数的使用与前面介绍的 `list()`、`get()` 等函数类似。此外，还可以使用 `QFtp` 类控制 FTP 的传输模式（主动或被动模式）和传输类型（二进制或字符类型），甚至通过 `rawCommand()` 函数执行任意 FTP 命令，如 `SITE CHMOD` 命令：

```
ftp->rawCommand("SITE CHMOD 755 kde");
```

读者可以根据应用需求进行选取。

9.2 HTTP 客户端

HTTP(Hypertext Transfer Protocol, 超文本传输协议)和 FTP 都是文件传送协议，它们有许多共同的特征，譬如说都运行在 TCP 之上。不过这两个应用层协议之间存在重要的差别。与 FTP 使用两个并行的 TCP 连接来传送文件不同，HTTP 只使用一个 TCP 连接，既用于承载请求和响应，也用于承载所传送的文件，如图 9-5 所示。

Qt 提供了一个 `QHttp` 类用于构建 HTTP 客户端程序，该类与前一节讲述的 `QFtp` 类有很多相似之处。它提供了许多常用的 HTTP 操作，如 `get()` 和 `post()` 函数，以及其他用于发送任意 HTTP 请求的方



法。QHttp 类也采用异步工作方式，当调用了 `get()` 或 `post()` 等函数后会立刻返回，而真正的数据传输直到程序控制权返回 Qt 事件循环后才会发生，从而加快了用户界面的响应。

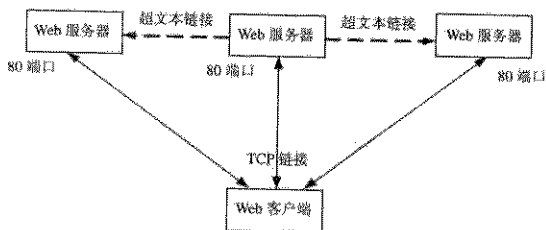


图 9-5 HTTP 模型

本节将展示一个 `wget` 的控制台示例，其功能与 Linux 下自带的 `wget` 命令类似，它可以使用 HTTP 协议从服务器端下载文件。

```

class HttpGet : public QObject
{
    Q_OBJECT
public:
    HttpGet(QObject *parent = 0);
    ~HttpGet();
    bool downloadFile(QUrl&);
signals:
    void done();
private slots:
    void httpRequestFinished(int requestId, bool error);
    void readResponseHeader(const QHttpResponseHeader &responseHeader);
    void updateDataReadProgress(int bytesRead, int totalBytes);
    void httpDone(bool error);
private:
    QHttp *http;
    QFile *file;
    int httpGetId;
    bool httpRequestAborted;
    QString fileName;
};
  
```

HttpGet 类与上节的 `FtpLogin` 类十分相似，它包含一个封装了到 HTTP 服务器端连接的 `QHttp` 型私有变量 `http`，一个用于记录下载操作 ID 的整型私有变量 `httpGetId`，一个表示操作是否终止的布尔型私有变量 `httpRequestAborted`，以及用于保存下载文件的 `QFile` 型私有变量 `file` 和记录下载文件名的 `QString` 型私有变量 `fileName`。下面介绍这个类的构造函数、槽函数和其他功能函数。构造函数负责创建 `QHttp` 实例并将四种信号与其相应的槽函数关联。

```

HttpGet::HttpGet(QObject *parent)
    : QObject(parent)
{
    http = new QHttp(this);
  
```

```

connect(http, SIGNAL(done(bool)), this, SLOT(httpDone(bool)));
connect(http, SIGNAL(requestFinished(int, bool)),
        this, SLOT(httpRequestFinished(int, bool)));
connect(http, SIGNAL(dataReadProgress(int, int)),
        this, SLOT(updateDataReadProgress(int, int)));
connect(http, SIGNAL(responseHeaderReceived(const QHttpResponseHeader
&)), this, SLOT(readResponseHeader(const QHttpResponseHeader &)));
}

```

构造函数负责创建 `QHttp` 实例并将四种信号与其相应的槽函数关联。信号 `done()`、`requestFinished()` 和 `dataReadProgress()` 信号与上一节介绍的 `QFtp` 类中的同名信号一样，分别在所有 HTTP 操作结束时，每一个 HTTP 操作结束时和从 HTTP 服务器读到数据时发生，而 `responseHeaderReceived()` 信号则在接到服务器端的 HTTP 响应头时发生。

```

bool HttpGet::downloadFile(QUrl& url)
{
    QFileInfo fileInfo(url.path());
    fileName = fileInfo.fileName();
    if (QFile::exists(fileName)) {
        qDebug() << tr("当前路径下文件%1 已存在。")
            .arg(fileName);
        httpRequestAborted = true;
        return false;
    }
    file = new QFile(fileName);
    if (!file->open(QIODevice::WriteOnly)) {
        qDebug() << tr("无法保存文件%1: %2.")
            .arg(fileName).arg(file->errorString());
        delete file;
        file = 0;
        httpRequestAborted = true;
        return false;
    }
    http->setHost(url.host(), url.port() != -1 ? url.port() : 80);
    if (!url.userName().isEmpty())
        http->setUser(url.userName(), url.password());
    httpGetId = http->get(url.path(), file);
    httpRequestAborted = false;
    return true;
}

```

函数 `downloadFile(QUrl&url)` 首先从 `main` 函数传入的 `QUrl` 型参数 `url` 中获取要下载的文件并将其文件名存储在私有成员 `fileName` 中，这里用到了 `QFileInfo` 类，它可以获取与平台无关的文件信息，如用 `fileName()` 函数剥去路径获得文件名；然后在本地当前目录下新建同名文件并以只写方式打开；接下来通过 `setHost()` 函数设置服务器主机名和地址，如果未指定端口则使用默认的 80 端口，如果请求的服务器需要认证，此时传入的 `url` 参数需包含用户名，程序员只需通过 `setUser()` 函数设置其用户名和相应密码就可以了；最后通过 `get()` 函数将文件下载到本地，同时在私有成员 `httpGetId` 中记录此次操作的操作 ID。如果操作成功该函数返回 `true` 并置 `httpRequestAborted` 为 `false`，表示请求成功，否则返

回 false, 同时置 httpRequestAborted 为 true, 表示请求终止。

```
void HttpGet::readResponseHeader(const QHttpResponseHeader &responseHeader)
{
    if (responseHeader.statusCode() != 200) {
        qDebug() << tr("下载失败: %1。")
            .arg(responseHeader.reasonPhrase());
        httpRequestAborted = true;
        http->abort();
        return;
    }
}
```

函数 readResponseHeader(const QHttpResponseHeader & responseHeader) 在得到服务器端响应时通过信号 responseHeaderReceived(const QHttpResponseHeader &) 调用, QHttpResponseHeader 型参数 responseHeader 包含了 HTTP 服务器的响应头部信息, 其功能函数 statusCode() 可以获取服务器端的响应码, 如最常见的 404 代码表示所请求的资源没找到, 这里查看其是否为 200 (请求成功), 如果不是, 则调用 QHttp 类的 abort() 函数终止操作, 同时置 httpRequestAborted 私有变量为 true, 表示请求终止。

```
void HttpGet::httpRequestFinished(int requestId, bool error)
{
    if (httpRequestAborted) {
        if (file) {
            file->close();
            file->remove();
            delete file;
            file = 0;
        }
        return;
    }
    if (requestId != httpGetId)
        return;
    file->close();
    if (error) {
        file->remove();
        qDebug() << tr("下载文件失败: %1")
            .arg(http->errorString());
    } else {
        qDebug() << tr("文件%1 下载完毕。").arg(fileName);
    }
    delete file;
    file = 0;
}
```

函数 httpRequestFinished(int, bool) 用来追踪 downloadFile() 函数中下载文件操作的执行情况。如果 httpRequestAborted 私有变量未表示请求终止, 当前操作号与下载操作中记载的 httpGetId 操作号相等, 且参数无出错标识, 就表示下载操作成功, 此时只需关闭本地文件并返回, 否则需要进行错误处理, 这里采取的措施是删除本地下载失败的文件并返回。

```
void HttpGet::updateDataReadProgress(int bytesRead, int totalBytes)
{
    if (httpRequestAborted)
        return;
    qDebug() << tr("已下载: %1%").arg(100*bytesRead/totalBytes);
}
```

函数 `updateDataReadProgress(int, int)` 用于更新当前下载文件的进度，这里以百分比的形式在控制台输出。

```
void HttpGet::httpDone(bool error)
{
    if (error) {
        qDebug() << tr("错误:") << qPrintable(http->errorString()) << endl;
    }
    emit done();
}
```

大多数情况下应用程序往往只需要知道全部操作序列何时完成，函数 `httpDone(bool)` 在结束所有 HTTP 操作后被 `QHttp` 类的 `done(bool)` 调用，并继续发出一个新的 `done()` 信号用于结束整个应用程序。

此外，`QHttp` 类还提供了 `post()`、`head()`、`request()` 等常用操作。如可以通过 `post()` 函数向 CGI 脚本发送某个名/值对，并将响应写入文件，操作如下所示：

```
http->setHost("www.example.com");
http->post("/cgi/somescript.py", "x=300&y=400",&file);
```

还可以通过以下方法构建任意 HTTP 头和数据，并通过请求函数将其发出。

```
QHttpRequestHeader header("GET", "/index.html");
header.setValue("Host", "www.trolltech.com");
http->setHost("www.trolltech.com");
http->request(header);
```

甚至可以为 `QHttp` 指定一个新的 socket 以取代其默认的 `QTcpSocket`，如使用 Qt 4 解决方案中提供的 `QtSslSocket` 类在 HTTP 协议中实现 SSL 加密功能（Qt 从 4.3 版本开始支持这项功能，此前则需要商业版授权，并安装 `qtsslsocket-2.5-commercial.tar.gz` 包）。

9.3 UDP 应用

UDP（User Data Protocol，用户数据报协议）是一种简单轻量级的传输层协议，它提供无链接的、不可靠的报文^①传输，非常适合下面 4 种情况。

- 网络数据大多为短消息
- 拥有大量客户端
- 对数据安全性无特殊要求
- 网络负担非常重，但对响应速度要求高

UDP 协议工作原理如图 9-6 所示。

① 报文是指有一定长度限制的数据包。



图 9-6 UDP 协议工作模型

客户端向服务器端发送一定长度的请求报文，报文大小的限制与各系统的协议实现有关，但不得超过其下层 IP 协议规定的 64K，如果服务器端未收到此请求，客户端不会进行重发，因此报文的传输是不可靠的，常用的聊天工具 QQ 就使用 UDP 协议发消息，因此有时会出现收不到消息的情况；服务器端可以同样以报文形式做出响应。

从本节开始，将介绍经典的 C/S（Client/Server，客户机/服务器）编程模型。基于 UDP 协议的客户机/服务器程序编写流程如图 9-7 所示。

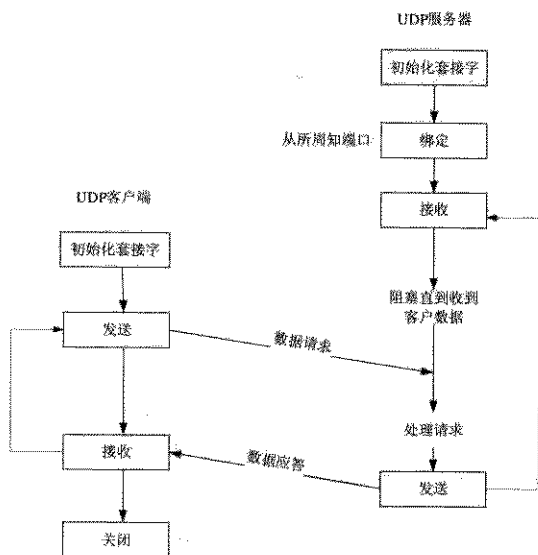


图 9-7 UDP 客户端与服务端间的交互时序

客户端不与服务器建立连接，它只管调用发送函数给服务器发送数据报。类似地，服务器也不从客户端接收连接，它只管调用接收函数，等待来自某客户端的数据到达。

Qt 提供了一个 `QUdpSocket` 类用于编写 UDP 程序，使用它可以很容易地实现数据报的收发。`QUdpSocket` 类提供的另一个重要功能是广播，该功能在发现网络资源时十分有用，但组播功能在 `QUdpSocket` 类中尚未得到支持，如何使用组播功能将在 9.5 节的高级应用中介绍。下面介绍一个简单的广播示例，它由客户端和服务端两部分组成。

客户端每隔一秒向子网内所有主机广播一次。


```

class Sender : public QObject
{
    Q_OBJECT
public:
    Sender(QObject *parent = 0);
    ~Sender();
    void start();
private slots:
    void broadcastDatagram();
private:
    QUdpSocket *udpSocket;
    QTimer *timer;
    int messageNo;
};

```

Sender 类从 QObject 类继承，并加入 Q_OBJECT 宏。QUdpSocket 型私有变量用来向服务器发送数据；定时器 QTimer 每隔一秒钟激活一次，同时进行一次广播；整型私有变量 messageNo 则用来对广播消息计数。

```

Sender::Sender(QObject *parent)
    : QObject(parent)
{
    timer = new QTimer(this);
    udpSocket = new QUdpSocket(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(broadcastDatagram()));
    messageNo = 1;
}

```

构造函数创建 QUdpSocket 实例和定时器实例，并关联定时器的超时信号 timeout() 到槽函数 broadcastDatagram()，然后为消息计数变量 messageNo 置初值 1。

```

void Sender::start()
{
    timer->start(1000);
}

```

函数 start() 负责启动定时器，它每隔一秒钟激活一次广播操作。

```

void Sender::broadcastDatagram()
{
    qDebug() << (tr("开始广播: %1").arg(messageNo));
    QByteArray datagram = "BroadCast Messages:" +
                           QByteArray::number(messageNo);
    udpSocket->writeDatagram(datagram.data(), datagram.size(),
                           QHostAddress::Broadcast, 44444);
    ++messageNo;
}

```

槽函数 broadcastDatagram() 实现了真正的数据发送，首先在 QByteArray 型局部变量 datagram 中构建待发送的数据包，然后通过 QUdpSocket 类的 writeDatagram (const char * data, qint64 size, const

`QHostAddress & address, quint16 port`)函数将数据包发出,该函数的4个参数依次是数据包自身、数据包大小、发送到的地址和端口。值得注意的是,这里的地址使用了 `QHostAddress::Broadcast` 值,它对应 IPv4 下的广播地址,如果将该值更换成单机地址(如本机地址 `QHostAddress::LocalHost`),本例将变成一个普通的端到端的 UDP 程序。

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    QTextCodec::setCodecForTr( QTextCodec::codecForName("gb18030"));
    Sender sender;
    sender.start();
    return app.exec();
}
```

`main()`函数十分简单,它只需要生成一个 `Sender` 实例并启动定时器,然后进入 Qt 事件循环。

服务器端程序的编写与客户端类似,所不同的仅是服务器端需要绑定事先约定好的端口,并读出接收到的数据。

```
class Receiver : public QObject
{
    Q_OBJECT
public:
    Receiver(QObject *parent = 0);
    ~Receiver();
private slots:
    void processPendingDatagrams();
private:
    QUdpSocket *udpSocket;
};
```

`Receiver` 类从 `QObject` 类继承,并加入 `Q_OBJECT` 宏。`QUdpSocket` 型私有变量用来接收客户端发送的数据包。

```
Receiver::Receiver(QObject *parent)
    : QObject(parent)
{
    udpSocket = new QUdpSocket(this);
    udpSocket->bind(44444);
    connect(udpSocket, SIGNAL(readyRead()),
            this, SLOT(processPendingDatagrams()));
}
```

构造函数生成 `QUdpSocket` 实例,并绑定与客户端约定的端口。需要注意的是,在编写网络程序时应避免使用 0 到 1023 端口,这些端口被称为公认端口,通常紧密绑定一些服务,如 80 和 21 端口分别用于上两节提到的 HTTP 和 FTP 服务,因此,为了避免冲突,在这里使用一个比较大的端口 44444。当接收到数据包时, `QUdpSocket` 会发出 `readyRead()` 信号,用来通知读取数据,这里将它与接收数据的槽函数 `processPendingDatagrams()` 关联。

```
void Receiver::processPendingDatagrams()
```

```

{
    while (udpSocket->hasPendingDatagrams()) {
        QByteArray datagram;
        datagram.resize(udpSocket->pendingDatagramSize());
        udpSocket->readDatagram(datagram.data(), datagram.size());
        qDebug() << (tr("接收数据: \"%1\"")
            .arg(datagram.data()));
    }
}

```

接收函数 `processPendingDatagrams()` 首先调用 `QUdpSocket` 类的公有成员函数 `hasPendingDatagrams()` 判断是否有可供读取的数据, 如果有则通过其另一个公有成员函数 `pendingDatagramSize()` 获取当前可供读取的 UDP 报文大小, 并据此大小分配接收缓冲区, 最后通过 `QUdpSocket` 类的 `readDatagram()` 函数将收到的报文读入接收缓冲区。

`main()` 函数与客户端类似, 这里不再赘述。在运行这个示例时, 如果服务器端收不到广播报文, 请首先关闭 Linux 下的防火墙。

现在完成了一个简单 C/S 结构的 UDP 应用程序, 客户端仅发送数据, 服务器仅负责接收。在实际应用中, 客户端和服务器双方均需要进行数据的收发, 也就是常说的全双工工作方式, 这时两端需分别使用 `QUdpSocket::bind()` 函数绑定各自的主机和端口来读取数据, 同时还可以在同一个套接字上使用 `QUdpSocket::writeDatagram()` 函数向不同的目的地址和端口发送数据。

9.4 TCP 应用

TCP (Transmission Control Protocol, 传输控制协议) 是一种面向连接和数据流的可靠传输协议。它是许多高层应用协议的基础, 如前面介绍的 FTP 和 HTTP 协议都是建立在 TCP 协议之上的。TCP 协议工作原理如图 9-8 所示。

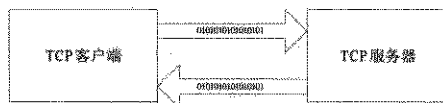


图 9-8 TCP 协议工作模型

TCP 协议能为应用程序提供可靠的通信连接, 使一台计算机发出的字节流无差错地发往网络上的其他计算机, 因此对可靠性要求高的数据通信系统往往使用 TCP 协议传输数据, 但在正式收发数据前, 通信双方必须建立连接。总结起来, TCP 协议与 UDP 协议的差别如表 9-1 所示。

表 9-1 TCP 协议与 UDP 协议的差别

	TCP	UDP
是否连接	面向连接	非面向连接
传输可靠性	可靠	不可靠
流量控制	提供	不提供
工作方式	全双工	可以是全双工
应用场合	大量数据	少量数据
速度	慢	快



基于 TCP 协议的客户端/服务器程序编写流程如图 9-9 所示。

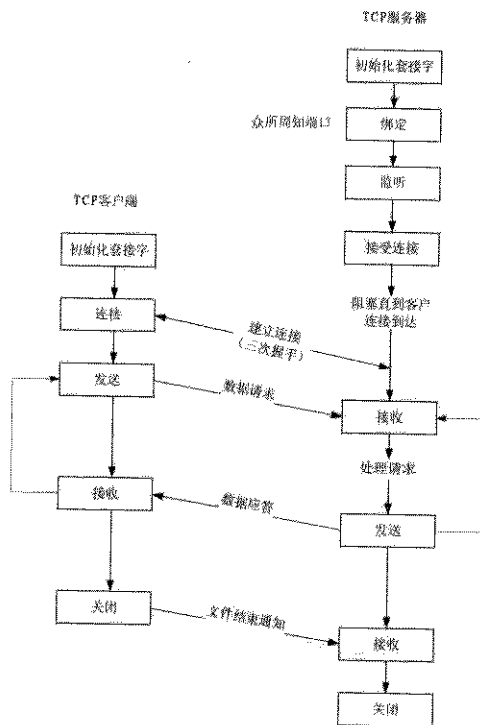


图 9-9 TCP 客户端与服务端间的交互时序

首先启动服务器，稍候的某个时刻启动客户端，它与此服务器经过三次握手后建立连接。此后的一段时间内，客户端向服务器发送一个请求，服务器处理这个请求，并且给客户发回一个响应。这个过程一直持续下去，直到客户端给服务器发一个文件结束符，并关闭客户端连接，接着服务器也关闭服务器端的连接，结束运行或等待一个新的客户端连接。

Qt 提供了 `QTcpSocket` 类和 `QTcpServer` 类用于编写 TCP 客户端和服务端应用程序。`QTcpSocket` 类提供了 TCP 协议的通用接口，可以用来实现其他标准协议，如 POP3、SMTP 和 NNTP 等，也可以用来实现自定义的协议。与 `QHttp`、`QFtp` 和 `QUdpSocket` 类相似，`QTcpSocket` 类也采用异步工作方式，它依靠 Qt 事件循环发现外来数据和向外发送的数据，并以信号的方式报告状态改变或产生的错误。由于 `QTcpSocket` 类通过其父类 `QAbstractSocket` 继承了 `QIODevice` 类，因此可以使用 `QTextStream` 和 `QDataStream` 这样的流结构类，从而大大方便了 TCP 数据流的读写。`QTcpServer` 类在服务器端处理外来的 TCP 客户端连接，需要注意的是，该类直接继承于 `QObject` 基类，而不是 `QAbstractSocket` 抽象套接字类。

下面介绍一个 TCP 应用示例，它由客户端和服务端两部分组成，客户端选择本地文件，并通过 TCP 连接将它上传到服务器端。由于使用了 TCP 协议，所以可以很轻松地传递大文件，而无需担心传



```

QPushButton *startButton;
QPushButton *quitButton;
QPushButton *openButton;
QDialogButtonBox *buttonBox;
QTcpSocket tcpClient;

qint64 TotalBytes;
qint64 bytesWritten;
qint64 bytesToWrite;
qint64 loadSize;
QString fileName;
QFile *localFile;
QByteArray outBlock;
};

```

Dialog 类中包含的最重要的成员是 tcpClient 私有变量,它是 QTcpSocket 的实例,封装了一条 TCP 连接。其余的成员变量起辅助作用,其中私有变量 localFile 和 fileName 用来记录待发送的文件与文件名。TotalBytes、bytesWritten 和 bytesToWrite 私有变量用来记录总共需发送的字节数、已发送字节数和待发字节数。为了发送较大的文件,这三个变量使用了 qint64 类型,Qt 保证该类型数据在所有其所支持的平台下均为 64 位大小,这几乎可以表示一个无限大的文件(打个形象的比方,如果每秒消耗 1GB 空间,64 位大小可表示的空间足够使用 500 多年)。outBlock 私有变量用来缓存一次发送的数据,而 loadSize 私有变量被初始化为一个 4KB 的常量,它用来尽可能地将一个较大的文件分割,每次发送 4KB 大小,余下不足 4KB 的按实际大小发送。用于界面显示的相关成员变量比较容易,这里省略。成员函数部分将在下面一一介绍。

```

Dialog::Dialog(QWidget *parent)
    : QDialog(parent)
{
    loadSize = 4*1024;
    TotalBytes = 0;
    bytesWritten = 0;
    bytesToWrite = 0;
    clientProgressBar = new QProgressBar;
    clientStatusLabel = new QLabel(tr("客户端就绪"));
    startButton = new QPushButton(tr("开始"));
    quitButton = new QPushButton(tr("退出"));
    openButton = new QPushButton (tr("打开"));
    startButton->setEnabled(false);

    .....

    connect(startButton, SIGNAL(clicked()), this, SLOT(start()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(openButton, SIGNAL(clicked()), this, SLOT(openFile()));
    connect(&tcpClient, SIGNAL(connected()), this, SLOT(startTransfer()));
    connect(&tcpClient, SIGNAL(bytesWritten(qint64)),
        this, SLOT(updateClientProgress(qint64)));
    connect(&tcpClient, SIGNAL(error(QAbstractSocket::SocketError)),
        this, SLOT(displayError(QAbstractSocket::SocketError)));

    .....
}

```

构造函数负责初始化界面,并将“开始”、“打开”和“退出”按钮与各自的槽函数关联。这里还关联了 `QTcpSocket` 的三个重要信号,它们分别是成功与服务器建立连接后产生的 `connected()` 信号,数据成功发送后产生的 `bytesWritten()` 信号和产生错误的 `error()` 信号。构造函数还将总发送字节数 `TotalBytes`、已发送字节数 `bytesWritten` 和待发字节数 `bytesToWrite` 置 0,同时约定每次的发送负载 `loadSize` 为 4KB。

```
void Dialog::openFile()
{
    fileName = QFileDialog::getOpenFileName(this);
    if (!fileName.isEmpty())
        startButton->setEnabled(true);
}
```

用户在客户端界面按下“打开”按钮后, `openFile()` 槽函数将被调用。该函数通过 Qt 文件选择对话框 `QFileDialog` 所提供的静态函数 `getOpenFileName()`,能够很容易地返回用户所选取的文件名,这里将其保存在私有成员变量 `fileName` 中。如果选中返回的文件名非空,将激活“开始”按钮。

```
void Dialog::start()
{
    startButton->setEnabled(false);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    bytesWritten = 0;
    clientStatusLabel->setText(tr("连接中..."));
    tcpClient.connectToHost(QHostAddress::LocalHost, 16689);
}
```

用户在客户端界面按下“开始”按钮后, `start()` 槽函数将被调用。该函数的主要功能是连接服务器,它使用了 `QTcpSocket` 类的 `connectToHost()` 函数,其中的两个参数分别是服务器主机地址及其监听端口,读者可以根据实际应用需求进行修改。

```
void Dialog::startTransfer()
{
    localFile = new QFile(fileName);
    if (!localFile->open(QFile::ReadOnly)) {
        QMessageBox::warning(this, tr("应用程序"),
            tr("无法读取文件 %1:\n%2.")
                .arg(fileName)
                .arg(localFile->errorString()));
        return;
    }
    TotalBytes = localFile->size();
    QDataStream sendOut(&outBlock, QIODevice::WriteOnly);
    sendOut.setVersion(QDataStream::Qt_4_3);

    QString currentFile = fileName.right(fileName.size() -
        fileName.lastIndexOf('/') - 1);
    sendOut << qint64(0) << qint64(0) << currentFile;
    TotalBytes += outBlock.size();
}
```



```

sendOut.device()->seek(0);
sendOut<<TotalBytes<<qint64((outBlock.size()-sizeof(qint64)*2));
bytesToWrite = TotalBytes - tcpClient.write(outBlock);
clientStatusLabel->setText(tr("已连接"));
qDebug()<<currentFile<<TotalBytes;
outBlock.resize(0);
}

```

一旦连接建立成功，QTcpSocket 类将发出 `connected()` 消息，继而调用 `startTransfer()` 槽函数。该函数首先向服务器端发送一个文件头结构。文件头结构由三个字段组成，分别是，64 位的总长度（包括文件数据长度和文件头自身长度），64 位的文件名长度和变长的文件名，如图 9-12 所示。

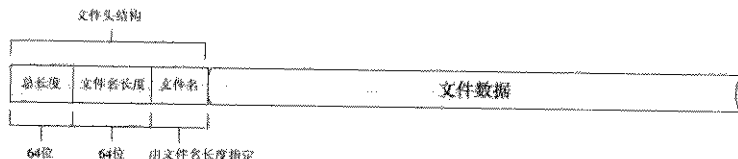


图 9-12 发送文件数据格式

函数 `startTransfer()` 首先以只读方式打开选中的文件，然后通过 `QFile` 类的 `size()` 函数获取待发送文件的大小，并将该值暂存于 `TotalBytes` 变量中。接下来将发送缓冲区 `outBlock` 封装在一个 `QDataStream` 类型的变量中，这样做可以很方便地通过重载的 “<<” 操作符填写文件头结构。为了使不同 Qt 版本可以互相读写经 `QDataStream` 类流化的数据，这里特别调用 `setVersion(QDataStream::Qt_4_3)` 函数设置流化数据格式，如果能够保证使用同一 Qt 版本，则可以省略该操作。

设置文件头结构的操作有些小技巧，这里首先通过 `QString` 类的 `right()` 函数去除文件的路径部分，仅将文件部分保存在 `currentFile` 变量中，然后通过 `sendOut<<qint64(0)<<qint64(0)<<currentFile` 操作构造一个临时文件头，并将总长度和文件名长度暂时置零。现在可以通过 `outBlock.size()` 函数获得文件头的实际存储大小，将该值追加到 `TotalBytes` 字段，从而完成实际需发送字节数的记录。接着通过 `sendOut.device()-> seek(0)` 函数将读写操作指向从头开始，并调用类似操作 `sendOut<<TotalBytes<<qint64((outBlock.size()-sizeof(qint64)*2))`，填写实际的总长度和文件长度。需要注意的是，不能错误地通过 `QString::size()` 函数获取文件名大小，该函数返回的是 `QString` 类型文件名所包含的字节数，而不是实际所占存储空间的大小，由于字节编码和 `QString` 类存储管理的原因，两者往往并不相等。

在完成了文件头结构的填写后，调用 `tcpClient.write(outBlock)` 函数将该文件头发出，同时修改待发送字节数 `bytesToWrite`。最后，调用 `outBlock.resize(0)` 函数清空发送缓冲区以备下次使用。

```

void Dialog::updateClientProgress(qint64 numBytes)
{
    bytesWritten += (int)numBytes;
    if (bytesToWrite > 0){
        outBlock = localFile->read(qMin(bytesToWrite, loadSize));
        bytesToWrite -= (int)tcpClient.write(outBlock);
        outBlock.resize(0);
    }
}

```



```

else{
    localFile->close();
}
clientProgressBar->setMaximum(TotalBytes);
clientProgressBar->setValue(bytesWritten);
clientStatusLabel->setText(tr("已发送 %1MB"
    .arg(bytesWritten / (1024 * 1024))));
}

```

一旦数据发出, `QTcpSocket` 类将会产生 `bytesWritten()` 信号, 继而调用 `updateClientProgress(qint64)` 槽函数, 参数表示实际已发出的字节数。如果待发送数据计数 `bytesToWrite` 大于 0, 将尽可能地从发送文件中读出 4KB 数据, 并将其发出, 否则发送完毕关闭文件。还需要在此更新已发和待发数据计数, 并以此更新发送进度条和状态显示。

```

void Dialog::displayError(QAbstractSocket::SocketError socketError)
{
    if (socketError == QTcpSocket::RemoteHostClosedError)
        return;

    QMessageBox::information(this, tr("网络"),
        tr("产生如下错误: %1.")
        .arg(tcpClient.errorString()));

    tcpClient.close();
    clientProgressBar->reset();
    clientStatusLabel->setText(tr("客户端就绪"));
    startButton->setEnabled(true);
    QApplication::restoreOverrideCursor();
}

```

如果连接或数据传输过程中的某次操作发生错误, `QTcpSocket` 类会发出 `error()` 信号, 并触发错误处理槽函数 `displayError()`。该函数的错误处理方式比较简单, 仅是显示出错对话框并关闭连接。因此, 本例的文件传输不提供断点续传功能, 如实际应用中有此需求, 读者可根据具体的错误提示进行处理。

`main()` 函数实现与前面的例子类似, 这里不再赘述。

服务器端程序 `ReceiveFile` 完成的功能与客户端程序恰恰相反, 它负责从 TCP 连接上接收数据, 并将其写入当前目录下的指定文件中。其界面也是一个简单的对话框, 上面布置了一个 `QProgressBar` 进度条, 一个用来显示状态的 `QLabel`, 两个 `QPushButton` 按钮分别用来开启监听和退出程序, 如图 9-10 所示。该程序的主要功能也是在一个从 `QDialog` 类继承而来的 `Dialog` 类中完成的, 下面将着重介绍。

```

class Dialog : public QDialog
{
    Q_OBJECT

public:
    Dialog(QWidget *parent = 0);

public slots:
    void start();
}

```



```

void acceptConnection();
void updateServerProgress();
void displayError(QAbstractSocket::SocketError socketError);

private:
    QProgressBar *clientProgressBar;
    QProgressBar *serverProgressBar;
    QLabel *serverStatusLabel;
    QPushButton *startButton;
    QPushButton *quitButton;
    QPushButton *openButton;
    QDialogButtonBox *buttonBox;

    QTcpServer tcpServer;
    QTcpSocket *tcpServerConnection;
    quint64 TotalBytes;
    quint64 bytesReceived;
    quint64 fileNameSize;
    QString fileName;
    QFile *localFile;
    QByteArray inBlock;
};

```

Dialog 类中包含的两个最重要成员是 tcpServer 和 tcpServerConnection 私有变量, 其中, tcpServer 变量是 QTcpServer 类的实例, 它负责监听并处理外来连接请求; tcpServerConnection 变量是 QTcpSocket 的实例, 它封装了一条 TCP 连接, 这个类在前面的客户端程序中已经介绍过。其余的成员变量起辅助作用, 其中私有变量 localFile 和 fileName 用来记录待接收的文件与文件名。TotalBytes、bytesWritten 和 bytesToWrite 私有变量用来记录总共需接收的字节数、已接收字节数和待接收字节数。界面显示的相关成员变量比较容易, 这里略过。成员函数部分将在下面一一介绍。

```

Dialog::Dialog(QWidget *parent)
    : QDialog(parent)
{
    TotalBytes = 0;
    bytesReceived = 0;
    fileNameSize = 0;
    serverProgressBar = new QProgressBar;
    serverStatusLabel = new QLabel(tr("服务端就绪"));
    startButton = new QPushButton(tr("接收"));
    quitButton = new QPushButton(tr("退出"));

    .....

    connect(startButton, SIGNAL(clicked()), this, SLOT(start()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(&tcpServer, SIGNAL(newConnection()),
           this, SLOT(acceptConnection()));

    .....
}

```

构造函数负责初始化界面，并将开始和退出按钮与各自的槽函数关联。这里还关联了 `QTcpServer` 的 `newConnection()` 信号，该信号在有可用的 TCP 连接时发出。

```
void Dialog::start()
{
    startButton->setEnabled(false);

    QApplication::setOverrideCursor(Qt::WaitCursor);
    bytesReceived = 0;

    while (!tcpServer.isListening() &&
           !tcpServer.listen(QHostAddress::LocalHost, 16689)) {
        QMessageBox::StandardButton ret = QMessageBox::critical(this,
            tr("循环"),
            tr("无法开始测试: %1."),
            .arg(tcpServer.errorString()),
            QMessageBox::Retry
            | QMessageBox::Cancel);
        if (ret == QMessageBox::Cancel)
            return;
    }
    serverStatusLabel->setText(tr("监听"));
}
```

当用户按下“接收”按钮后，`start()` 函数开始执行，它调用 `QTcpServer` 的 `isListening()` 函数和 `listen()` 函数判断当前服务器是否已处在监听状态以及在本机 16689 端口建立监听是否成功。如果一切正常，服务器端就已经成功监听，随时等待处理客户端的 TCP 连接请求，否则打印出错消息，报告错误后返回。

```
void Dialog::acceptConnection()
{
    tcpServerConnection = tcpServer.nextPendingConnection();
    connect(tcpServerConnection, SIGNAL(readyRead()),
        this, SLOT(updateServerProgress()));
    connect(tcpServerConnection, SIGNAL(error(QAbstractSocket::SocketError)),
        this, SLOT(displayError(QAbstractSocket::SocketError)));
    serverStatusLabel->setText(tr("接受连接"));
    tcpServer.close();
}
```

当有客户端连接请求到来时，`QTcpSocket` 类将会发出 `newConnection()` 信号，从而触发 `acceptConnection()` 函数。`QTcpServer` 类在接受了外来 TCP 连接请求后，可以通过 `nextPendingConnection()` 函数获取一个新的已建立连接的子套接字，（该套接字封装在 `QTcpSocket` 类中）并返回 `QTcpSocket` 类指针，将返回值保存在 `tcpServerConnection` 私有变量中。接下来关联 `QTcpSocket` 类的 `readyRead()` 信号和 `error()` 信号，其中 `readyRead()` 信号在新连接中有可读数据时发出，而当新连接中产生错误时会发出 `error()` 信号。由于本例只处理一个客户端请求，因此在返回一个连接后，就调用 `QTcpSocket` 类的 `close()` 函数关闭服务器端的监听，后面的工作均在新建立的 `tcpServerConnection` 连接上完成。

```
void Dialog::updateServerProgress()
{

```

```

QDataStream in(tcpServerConnection);
in.setVersion(QDataStream::Qt_4_3);
if(bytesReceived <= sizeof(qint64)*2){
    if((tcpServerConnection->bytesAvailable() >=
        sizeof(qint64)*2)&&(fileNameSize ==0)){
        in>>TotalBytes>>fileNameSize;
        bytesReceived += sizeof(qint64)*2;
    }
    if((tcpServerConnection->bytesAvailable() >=
        fileNameSize)&&(fileNameSize !=0)){
        in>>fileName;
        bytesReceived += fileNameSize;
        localFile = new QFile(fileName);
        if (!localFile->open(QFile::WriteOnly)) {
            QMessageBox::warning(this, tr("应用程序"),
                tr("无法读取文件 %1:\n%2."),
                    .arg(fileName)
                    .arg(localFile->errorString()));
            return;
        }
    }
    else{
        return;
    }
}
if (bytesReceived < TotalBytes){
    bytesReceived += tcpServerConnection->bytesAvailable();
    inBlock = tcpServerConnection->readAll();
    localFile->write(inBlock);
    inBlock.resize(0);
}
serverProgressBar->setMaximum(TotalBytes);
serverProgressBar->setValue(bytesReceived);
qDebug()<<bytesReceived;
serverStatusLabel->setText(tr("已接收 %1MB")
    .arg(bytesReceived / (1024 * 1024)));
if (bytesReceived == TotalBytes) {
    tcpServerConnection->close();
    startButton->setEnabled(true);
    QApplication::restoreOverrideCursor();
}
}
}

```

当建立的连接有新的可供读取的数据时，QTcpSocket 类会发出 readyRead() 信号，从而触发 updateServerProgress() 函数。该函数完成数据的接收、存储，并更新进度显示。

首先将上面返回的 TCP 连接 tcpServerConnection 封装在 QDataStream 类型变量 in 中，同时设置流化数据格式类型为 QDataStream::Qt_4_3，与客户端保持一致。现在可以很方便地通过重载后的“<<”操作符读取 TCP 连接上的数据了。

由于流数据是没有结构的，为了知道接收的文件名以及文件何时接收完毕，必须首先获得文件头

结构(如图 9-12 所示)。这里还有一个小问题,由于开始时所传输文件名的长度是未知的,导致文件头结构的长度也是未知的,因此无法知道 TCP 数据流中前多少字节属于文件头结构部分。实际上文件头结构的接收分两步完成。

第一步是从 TCP 数据流中接收前 16 个字节(两个 qint64 结构长),用来确定总共需接收的字节数和文件名长度,并将这两个值保存在私有成员 TotalBytes 和 fileNameSize 中,然后根据 fileNameSize 值接收文件名。值得注意的是,无法保证在上述接收文件头结构的过程中,TCP 连接上总是有足够的数,因此,在第一步中,需要通过(tcpServerConnection->bytesAvailable() >= sizeof(qint64)*2)&&(fileNameSize == 0)操作确保至少有 16 字节的可用数据且文件名长度为 0(表示未从 TCP 连接接收文件名长度字段,仍处于第一步操作中),然后调用 in>>TotalBytes>>fileNameSize 操作读取总共需接收的数据和文件名长度。

第二步中类似地通过(tcpServerConnection->bytesAvailable() >= fileNameSize)&&(fileNameSize != 0)操作确保连接上的数据已包含完整的文件名且文件名长度不为 0(表示已从 TCP 连接接收文件名长度字段,处于第二步操作中),然后调用 in>>fileName 操作读取文件名,并根据该文件名在本地以只写方式打开一个同名文件 localFile,用来保存接收到的数据。

接下来的工作是读取实际的文件数据并保存,以及更新进度显示,直到收到全部数据。由于所发送的文件内容自身也是无格式的流,因此在接收文件内容时,只要 TCP 连接上有数据,就调用 tcpServerConnection->readAll()操作将当前全部可读数据读入接收缓冲 inBlock 中,随后再将该缓冲中的数据写入文件 localFile 中。当已收到的数据 bytesReceived 等于总数据 TotalBytes 时,接收完毕,这时通过 tcpServerConnection->close()操作关闭连接。最后,错误处理槽函数 displayError()和主函数 main()与客户端程序类似,这里不再赘述。

此外,需要提醒读者注意的是,通常 QTcpSocket 类和 QTcpServer 类以异步方式工作,但可以通过调用其 waitFor...()类型的函数实现同步操作,这类操作将阻塞调用线程直到某个信号发出。例如,在调用了非阻塞的 QTcpSocket::connectToHost()函数后紧接着调用 QTcpSocket::waitForConnected()函数以阻塞调用线程,直到 connected()信号发出。一般而言,同步操作往往可以简化代码的控制流程,但也存在较大的缺点,调用 waitFor...()函数将阻塞事件的处理,对于 GUI 线程会引起用户界面的冻结。因此,Qt 建议仅在非 GUI 线程中使用同步套接字,此时 QTcpSocket 也不再需要事件循环。

9.5 高级应用

Qt 网络模块依靠其特有的事件循环机制实现了异步模式的网络编程(该模式类似 MFC 中的异步 Socket),这意味着程序员在单一的 GUI 线程中直接使用或继承使用 Qt 网络模块中所提供的类时不会影响界面操作等其他功能。但凡事有利必有弊,Qt 网络模块在提供便利编程模式的同时,与直接使用操作系统套接字相比牺牲了一定的效率,这种效率损失通常情况下可以忽略不计,只有在特定情况下需要特别关注;另一方面,Qt 网络模块提供的操作不够全面,尤其是底层操作,例如,它目前尚无法直接提供组播功能,虽然这一点在 4.3 版本中有所改善,但仍然不够理想。本节将讨论这些问题,并给出一些解决办法。

9.5.1 底层操作

早期 Qt 版本所提供的底层网络操作功能几乎为零,幸好 TrollTech 公司已经意识到了这个问题,从 Qt 4.2 版开始提供了 QNetworkInterface 和 QNetworkAddressEntry 两个类用于操作底层网络接口。程



序员如果希望编写移植性好的网络程序，通常不直接在编码中写入本机地址、掩码等网络接口信息，而是先查询本地网络接口，然后根据查询结果选择相关信息用于网络程序的编写。下面使用 Qt 4.2 开始提供的 `QNetworkInterface` 类和 `QNetworkAddressEntry` 类模仿 Linux 的 `ifconfig -a` 命令编写示例程序 `showifconfig`。

当在 Linux 命令行下敲入 `ifconfig -a` 命令后，系统会显示所有的本机网络接口信息，如图 9-13 所示。

```
eth0      Link encap:Ethernet  HWaddr 00:0C:29:1B:FF:61
          inet addr:192.168.2.5   Bcast:192.168.2.255
          Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe1b:ff61/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500
          Metric:1
          RX packets:13005 errors:0 dropped:0 overruns:0 frame:0
          TX packets:308470 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4197119 (4.0 MiB)  TX bytes:431381347 (411.3
          MiB)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1   Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:2826 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2826 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4878330 (4.6 MiB)  TX bytes:4878330 (4.6 MiB)
          .....
          RX bytes:431381347 (411.3 MiB)  TX bytes:4233832 (4.0
          MiB)
xenbr0    Link encap:Ethernet  HWaddr FE:FF:FE:FF:FE:FF
          inet6 addr: fe80::200:ff:fe00:0/64 Scope:Link
          UP BROADCAST RUNNING NOARP  MTU:1500
          Metric:1
          RX packets:308059 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:426768990 (406.9 MiB)  TX bytes:0 (0.0 b)
```

图 9-13 本机网络接口信息

示例程序 `showifconfig` 只显示了网络接口个数、接口名、IP 地址、掩码等网络编程常用的基本信息，以及登录用户名。

```
int main()
{
    QStringList envVariables;
    QByteArray username;
    QList<QHostAddress> broadcastAddresses;
    QList<QHostAddress> ipAddresses;
    envVariables << "USERNAME.*" << "USER.*" << "USERDOMAIN.*"
```

```

    << "HOSTNAME.*" << "DOMAINNAME.*";
QStringList environment = QProcess::systemEnvironment();
foreach (QString string, envVariables) {
    int index = environment.indexOf(QRegExp(string));
    if (index != -1) {
        QStringList stringList = environment.at(index).split("=");
        if (stringList.size() == 2) {
            username = stringList.at(1).toUtf8();
            qDebug() << username.data();
            break;
        }
    }
}

broadcastAddresses.clear();
ipAddresses.clear();
qDebug() << "Interface numbers"
    << QNetworkInterface::allInterfaces().count();
foreach (QNetworkInterface interface,
        QNetworkInterface::allInterfaces()) {
    qDebug() << "Interface name:" << interface.name() << endl
        << "Interface hardwareAddress:"
        << interface.hardwareAddress() << endl
        << "entry numbers:" << interface.addressEntries().count();
    foreach (QNetworkAddressEntry entry, interface.addressEntries()) {
        QHostAddress broadcastAddress = entry.broadcast();
        qDebug() << "entry ip:" << entry.ip()
            << "entry netmask:" << entry.netmask();
        if (broadcastAddress != QHostAddress::Null) {
            broadcastAddresses << broadcastAddress;
            ipAddresses << entry.ip();
        }
    }
}
}
}

```

该程序第一步在系统环境变量中检索用户名，使用的方法是用正则表达式匹配环境变量中任何以 USERNAME 或 USER 或 USERDOMAIN 或 HOSTNAME 或 DOMAINNAME 开头的字符串，发生匹配时等号右侧的值即为用户名。需要注意获取系统环境变量的函数 `QProcess::systemEnvironment()` 是 Qt 4.1 版开始引入的一个静态函数，该函数以名/值对 (key=value) 列表的形式返回系统环境变量信息；另一个需提请读者注意的是用于循环控制的 `foreach()` 语句，该语句是 Qt 对标准 `for()` 循环结构的扩展，能够方便地实现对容器中所有项的遍历。

第二步工作是查询本机所有的网络接口信息。`QNetworkInterface` 类的 `allInterfaces()` 静态函数可以用来列出主机所有网络接口，每一个网络接口可以包含零个或多个 IP 地址项，以及与之对应的掩码和广播地址。可以调用 `addressEntries()` 函数获取地址项 `QNetworkAddressEntry` 类的列表，接下来可以调用 `QNetworkAddressEntry` 类的 `ip()`、`netmask()` 和 `broadcast()` 函数分别获取对应的 IP 地址、掩码和广播信息。如果用户只需要查询 IP 地址，则直接调用 `QNetworkInterface` 类的 `allAddresses()` 静态函数，它将以 `QHostAddress` 类列表的形式返回本机所有 IP，从而省去了对每一个地址项类



QNetworkAddressEntry 的操作。

9.5.2 使用代理

从 Qt 4.1 起提供了一个网络层的代理类 QNetworkProxy。该类为其他 Qt 网络类提供配置网络层代理的方法，目前支持的类包括 QAbstractSocket、QTcpSocket、QUdpSocket、QTcpServer、QHttp 和 QFtp。因此，本章介绍的所有网络程序都自动支持网络代理，使用步骤如下：

```
QNetworkProxy proxy;
proxy.setType(QNetworkProxy::Socks5Proxy);
proxy.setHostName("proxy.example.com");
proxy.setPort(1080);
proxy.setUser("username");
proxy.setPassword("password");
QNetworkProxy::setApplicationProxy(proxy);
```

每一种代理类型都遵守一定的约束，选择代理类型需详细查看 ProxyType 文档。目前 Qt 4 所支持的 SOCKS5 类型代理基于 RFC1928 和 RFC1929 规范。

另一种设置应用范围代理的方法是使用 QAbstractSocket::setProxy() 和 QTcpServer::setProxy() 为单独的套接字设置代理。如果需要关闭代理可使用如下方法。

```
QTcpServer serverSocket
serverSocket->setProxy(QNetworkProxy::NoProxy);
```

9.5.3 扩展 Qt 网络功能

Qt 提供的各种网络操作类使用虽然十分方便，但均缺乏对底层套接字的操作支持。幸运的是 Qt 网络模块的抽象基类 QAbstractSocket 提供了一对获取和设置套接字的操作，socketDescriptor() 与 setSocketDescriptor() 函数。下面将通过使用这两个函数并结合系统函数，演示一个组播通信的例子。

组播是指一个报文向一个“主机组”的传送，这个包含零个或多个主机的主机组由一个单独的 IP 地址标识。主机组地址称为“组播地址”，或者 D 类地址，这类 IP 地址的最高四位为“1110”，范围从 224.0.0.0 到 239.255.255.255。除了目的地址部分，组播报文与普通报文没有区别，网络尽力传送组播报文但并不保证一定送达。主机组的成员可以动态变化，主机有权选择加入或退出某个主机组。主机可以加入多个主机组，也可以向自己没有加入的主机组发送数据。

实际上对程序员而言，组播与普通的 UDP 应用并无太大区别，所不同的主要是在接收方，它需把用于接收的数据报型套接字加入特定的组播组，而在发送方，大多数情况下无需特别处理。下面首先介绍服务器端的接收部分。

由于 Qt 的 QUdpSocket 类未能提供加入组播组的相关操作，所以程序不得不使用操作系统提供的底层函数来完成该功能。为此，定义了 CBase.h 头文件，该文件包含了所有可能引用的操作系统头文件，并定义了一些数据类型和宏以备扩展使用。而接收端的主要功能由 GroupServer 类完成。

```
class GroupServer : public QObject{
    Q_OBJECT
public:
    GroupServer(){fSocket = 0;udpSocket = 0;count=0;Receivedpackets=0;};
    ~GroupServer(){};
```



```

void initSocket():
public slots:
    void readPendingDatagrams();
private:
    QUdpSocket * udpSocket;
    unsigned long FSocket;
    long count, Receivedpackets;
};

```

GroupServer 从 QObject 类派生,并在定义中使用了 Q_OBJECT 宏。其网络操作功能由私有成员变量 udpSocket 提供,另一个重要的私有成员 FSocket 用于记录将进行底层操作的套接字句柄。私有成员 count 和 Receivedpackets 分别用来记录接收报文编号和实际接收报文数。功能函数 initSocket() 实现了加入组播组的操作,而槽函数 readPendingDatagrams() 与前面介绍的功能一样,用来读取报文。构造函数十分简单,这里直接对 4 个私有成员赋初值。

```

void GroupServer::initSocket()
{
    unsigned long FSocket = socket(PF_INET, SOCK_DGRAM, IPPROTO_IP);
    struct ip_mreq imreq;
    imreq.imr_multiaddr.s_addr = inet_addr("225.1.0.111");
    imreq.imr_interface.s_addr = htonl(INADDR_ANY);
    setsockopt(FSocket, IPPROTO_IP, IP_ADD_MEMBERSHIP,
        (char *) &imreq, sizeof(ip_mreq));
    struct sockaddr_in FAddr;
    memset(&FAddr, 0, sizeof(FAddr));
    FAddr.sin_family = AF_INET;
    FAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    FAddr.sin_port = htons(8999);
    bind(FSocket, (sockaddr*)&FAddr, sizeof(FAddr));
    udpSocket = new QUdpSocket();
    udpSocket->setSocketDescriptor(FSocket);
    connect(udpSocket, SIGNAL(readyRead()),
        this, SLOT(readPendingDatagrams()));
}

```

函数 initSocket() 是本节讲述的重点,它使标准的 QUdpSocket 类实例加入了一个特定的组播组“225.1.0.111”。与以往的程序相比,这段代码比较晦涩难懂,这主要是因为用到了大量的操作系统套接字操作,下面将对此一一解释。实际的操作分三步进行。第一步操作创建一个套接字并将其加入组。首先使用系统函数 socket() 创建一个数据报型的套接字(通过参数 SOCK_DGRAM),并将返回的句柄保存在私有变量 FSocket 中以备后面引用。接下来用到了系统的一个结构 struct ip_mreq, 定义如下:

```

struct ip_mreq
{
    struct in_addr imr_multiaddr; /* IP multicast address of group */
    struct in_addr imr_interface; /* local IP address of interface */
};

```

成员 imr_multiaddr 用来记录组播地址,另一个成员 imr_interface 用来记录需加入组的网络接口 IP,这里使用 INADDR_ANY 值,表示由操作系统自动填上它所在机器的 IP 地址。最后使用 setsockopt() 系统函数将所创建的套接字加入组(通过参数 IP_ADD_MEMBERSHIP 表示本次设置套接字为加入组



操作)。第二步操作将这个已加入组的套接字绑定到端口 8999, 这里用到了底层的地址结构 `struct sockaddr_in` 和 `bind()` 函数。最后一步创建一个 `QUdpSocket` 实例, 并使用 `setSocketDescriptor()` 函数将其套接字句柄替换成 `FSocket`。至此, 该实例已具备接收组播的能力。

需要提请读者注意是, 不能先创建 `QUdpSocket` 实例, 然后使用 `socketDescriptor()` 函数获取套接字句柄, 最后据此句柄进行组播设置。原因在于此时 `QUdpSocket` 类处于非连接状态, `socketDescriptor()` 将返回一个非法句柄值。

```
void GroupServer::readPendingDatagrams()
{
    while (udpSocket->hasPendingDatagrams()) {
        QByteArray datagram;
        datagram.resize(udpSocket->pendingDatagramSize());
        QHostAddress sender;
        quint16 senderPort;
        udpSocket->readDatagram(datagram.data(), datagram.size(),
                                &sender, &senderPort);
        memcpy(&count, datagram.data(), sizeof(long));
        qDebug() << count << "----->>>" << "Received packets";
        Receivedpackets++;
    }
}
```

槽函数 `readPendingDatagrams()` 当有数据报到时通过信号 `readyRead()` 调用, 这里只是简单的打印报文编号和收到报文的计数。

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    GroupServer* serv = new GroupServer();
    serv->initSocket();
    QPushButton *button = new QPushButton("Quit");
    QObject::connect(button, SIGNAL(clicked()), &app, SLOT(quit()));
    button->show();
    return app.exec();
}
```

主函数比较简单, 只是启动了一个 `GroupServer` 实例并进行初始化, 然后使用一个退出按钮来控制程序退出。

发送端方面与使用 `QUdpSocket` 类发送普通的 UDP 报文一样, 通常不需要做出任何修改, 但如果考虑跨网段发送就需要做相应设置。IP 组播分组在互联网上的转发由支持组播的路由设备来处理。主机发出的 IP 组播分组在本子网内被所有主机组成员接收, 同时支持组播的路由器会把组播报文转发到所有包含该主机组成员的相邻网段上。组播报文传递的范围由报文的生存周期值 (TTL, Time-to-Live) 决定。如果 TTL 值等于或小于设置的路由器端口 TTL 门限值 (TTL Threshold), 路由器将不再转发该报文。这里要做的工作就是设置 TTL 值。

```
class GroupSend : public QObject{
    Q_OBJECT
public:
```

```

    GroupSend();
    ~GroupSend();
    void initSocket();
public slots:
    void sendPakages();
private:
    QUdpSocket * udpSocket;
    char* FSendBuffer;
    QTimer* timer;
    unsigned long FSocket;
    long count;
};

```

为此，我们定义了一个发送组播的类 **GroupSend**。该类使用一个定时器每 100 毫秒将缓冲区 **FSendBuffer** 中的内容向外发送一次。

```

GroupSend::GroupSend()
{
    count = 0;
    FSendBuffer = (char*)malloc(sizeof(long)+1024*16);
    memcpy(FSendBuffer, &count, sizeof(long));
    timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(sendPakages()));
}

```

构造函数开辟了一个 16KB+4 字节大小的发送缓冲区，其中前 4 字节为发送报文编号，然后新建一个定时器。

```

void GroupSend::initSocket()
{
    unsigned long FSocket = socket(PF_INET, SOCK_DGRAM, IPPROTO_IP);
    unsigned char TTL = 16;
    bool Loop = true;
    setsockopt(FSocket, IPPROTO_IP, IP_MULTICAST_TTL, (const char *)
        &TTL, sizeof(unsigned char));
    setsockopt(FSocket, IPPROTO_IP, IP_MULTICAST_LOOP, (const char *)
        &Loop, sizeof(unsigned char));
    udpSocket = new QUdpSocket();
    udpSocket->setSocketDescriptor(FSocket);
    timer->start(100);
}

```

初始化函数 **initSocket()** 使用系统函数 **socket()** 创建一个数据报类型的套接字，然后两次调用 **setsockopt()** 函数对该套接字进行设置。第一次使用 **IP_MULTICAST_TTL** 参数，将 **TTL** 值设置为 16，第二次使用 **IP_MULTICAST_LOOP** 参数，使组播报文环路有效。最后用这个套接字替换 **QUdpSocket** 实例中原有的套接字，并启动定时器。

```

void GroupSend::sendPakages()
{
    qDebug()<<tr("packages sent:")<<count;
    udpSocket->writeDatagram(FSendBuffer, sizeof(long)+1024*16,
        QHostAddress ("225.1.0.111"), 8999);
}

```



```

count++;
memcpy(FSendBuffer, &count, sizeof(long));
}

```

函数 `sendPakages()` 每一毫秒被调用一次, 它向组播地址 225.1.0.111 发送一个 16KB+4 字节大小的报文, 并在报文的前 4 字节从 0 开始顺序编号。

```

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    GroupSend* send = new GroupSend();
    send->initSocket();
    QPushButton *button = new QPushButton("Quit");
    QObject::connect(button, SIGNAL(clicked()), &app, SLOT(quit()));
    button->show();
    return app.exec();
}

```

主函数比较简单, 只是启动了一个 `GroupSend` 实例并进行初始化, 然后使用一个退出按钮来控制程序退出。

9.5.4 效率问题

Qt 网络模块的效率总体上还是令人满意的, 但当网络事件十分频繁时, 用户的界面操作将淹没于网络事件的汪洋大海之中, 从而造成用户界面冻结无法响应。这种情况在任何程序设计中都是不可接受的。上一节的例子是作者对 Qt 网络模块效率进行测试的一个简单用例, 有兴趣的读者可以加快客户端的发送频率, 直到接收端界面失去响应。在这个例子中, 可能还会发生数据包丢失的现象 (一旦产生数据包丢失, 接收端打印出的报文编号与接收报文数将不相等), 该现象在 UDP 报文传输时无法避免, 但合理地分配网络负载可以减轻这种状况。

为了提高网络效率, 可以采取以下三种措施。第一种方法在事件处理一章中将有介绍, 可以在 GUI 线程中适当调用 `QApplication::processEvents()` 方法, 它通知 Qt 来处理任何没有被处理的事件, 并且将控制权返回给调用者, 从而增加了用户界面操作的响应几率。第二种方法是启用新的线程处理网络事件, 从而使 GUI 线程可以用来专门处理用户界面操作, 该方法将在线程一章进行详细介绍。最后一种方法是完全撤开 Qt 网络模块, 直接使用操作系统套接字编程, 这种方法可以最大限度地提高效率并具有最大限度的灵活性。但是, 这种编程方法比较复杂, 对程序员的能力有一定要求, 已超出了本章的讨论范围, 有兴趣的读者可以参考 W.Richard Stevens 著, 施振川等译的经典著作《UNIX 网络编程第 1 卷: 套接口 API 和 X/Open 传输接口 API》。

9.6 小 结

这一章由浅入深地依次介绍了使用 Qt 网络模块开发 FTP、HTTP、TCP、UDP 应用程序 (Qt 并未提供网络层和链路层的相关接口, 但这两层在实际应用中也很少用到, 因此本章也略过了这部分内容), 并在最后一节介绍了查询主机、接口、以及使用代理和如何扩展 Qt 网络功能等高级话题。为了使读者更好的理解网络编程, 本章不仅穿插介绍了必要的网络和协议知识, 还客观地评价了 Qt 的网络功能, 并提出了使用建议, 希望能给读者在日后的开发中合理选择技术手段提供帮助。

第 10 章 多线程

在前面的章节中所看到的应用程序都是在唯一的图形用户界面 (GUI) 线程中每次执行一个操作。这种方式简单易行, 在大多数情况下可以满足应用需求, 但正如前面看到的, 当调用一个耗时操作 (如大批量 I/O 或大量矩阵变换等 CPU 密集操作) 时, 用户界面常常会冻结。第 11 章事件处理对此问题将提供一些解决方法, 本章将提供另一种解决方法。

早在 20 世纪 60 年代就已经提出了线程技术, 但真正将多线程应用到操作系统中是在 20 世纪 80 年代中期。传统的 UNIX 系统也支持线程的概念, 但在一个进程中只允许有一个线程, 这样多线程实际上就是多进程。Linux 下的 Posix 线程 (pthreads) 是一种轻量级进程的移植性实现。在 Linux 系统中, 线程的调度是由内核来完成的, 每个线程都有自己的编号, 由于在使用线程的软件项目中, 总体消耗的系统资源比较少, 加之线程间相互通信也比较容易, 因此在工程实践中推荐使用线程。多线程具有以下几点优势:

(1) 提高应用程序的响应速度。这对于开发图形界面的程序尤其重要, 当一个操作耗时很长时, 整个系统都会等待这个操作。程序就不能响应键盘、鼠标、菜单等的操作, 而使用多线程技术可将耗时的操作置于一个新的线程, 从而避免上述问题。

(2) 使多 CPU 系统更加有效。当线程数不大于 CPU 数目时, 操作系统可以调度不同的线程运行于不同的 CPU 上。

(3) 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程, 成为独立或半独立的运行部分, 这样有利于程序的理解和维护。

10.1 启动一个线程

在 Qt 中使用线程十分简单, 只需要继承 `QThread` 类并重新实现其 `run()` 函数, 代码如下所示。

```
class MyThread : public QThread
{
    Q_OBJECT
protected:
    void run();
};
void MyThread::run()
{
    ...
}
```

只需在 `run()` 函数中填写所需的功能代码, 然后创建一个 `MyThread` 实例, 并以 `QThread::start()` 函数启动这个实例即可。这样 `run()` 函数中的功能代码就运行在一个独立的线程中了。

接下来看一个具体的线程启动实例, 如图 10-1 所示。按下“开始”按钮将启动数个工作线程 (工



作线程数目由 MAXSIZE 宏决定), 各线程循环打印数字 0~9, 直到按下“停止”按钮终止所有线程。



图 10-1 线程启动示例

```
class WorkThread : public QThread
{
protected:
    void run();
};
void WorkThread::run()
{
    while(true)
        for (int n = 0; n < 10; ++n) {
            printf("%d%d%d%d%d%d%d%d\n", n, n, n, n, n, n, n, n);
        }
}
```

工作线程 `WorkThread` 从 `QThread` 继承而来, 这里重新实现其 `run()` 函数。`run()` 函数实际上是一个死循环, 它一刻不停地打印数字 0~9。为了显示效果明显, 程序将每一个数字重复打印 8 次。这里有一点经验供读者分享, 线程会因为调用 `printf()` 而持有一个控制 I/O 的锁 (lock), 多个线程同时调用 `printf()` 函数在某些情况下会造成控制台输出阻塞, 而使用 Qt 提供的 `qDebug()` 函数作为控制台输出一般不会出现上述问题。

```
#define MAXSIZE 5
class ThreadDlg : public QDialog
{
    Q_OBJECT
public:
    ThreadDlg(QWidget *parent = 0);
public slots:
    void start();
    void stop();
private:
    QPushButton *startButton;
    QPushButton *quitButton;
    QPushButton *stopButton;
    QDialogButtonBox *buttonBox;
    WorkThread* threadVector[MAXSIZE];
};
```

`ThreadDlg` 类提供了一个指向工作线程的私有指针数组 `threadVector`, 它记录了所启动的全部线程, 公共槽函数 `start()` 和 `stop()` 分别用于启动和终止线程, 线程数目由 `MAXSIZE` 宏定义。余下的私有变量用于界面显示, 它们在构造函数中完成初始化工作, 这里全部省略了, 详情请参考实现代码。

```
void ThreadDlg::start()
```

```

{
    for(int i=0;i<MAXSIZE;i++)
    {
        threadVector[i] = new WorkThread();
    }
    for(int i=0;i<MAXSIZE;i++)
    {
        threadVector[i]->start();
    }
    stopButton->setEnabled(true);
    startButton->setEnabled(false);
}

```

当用户按下“开始”按钮时，槽函数 start() 将被调用，它创建指定数目的 WorkThread 线程，并将 WorkThread 实例的指针保存在指针数组 threadVector 中。接下来调用 QThread 基类的 start() 函数，该函数将启动函数 run()，从而使得线程开始真正运行。这里使用了两个 while 循环，目的是为了新建的线程尽可能同时开始执行。

```

void ThreadDlg::stop()
{
    for(int i=0;i<MAXSIZE;i++)
    {
        threadVector[i]->terminate();
        threadVector[i]->wait();
    }
    startButton->setEnabled(true);
    stopButton->setEnabled(false);
}

```

当用户按下“停止”按钮时，stop() 槽函数将被调用，它依次终止保存在 threadVector 数组中的 WorkThread 实例，方法是调用 QThread 基类的 terminate() 函数。但是 terminate() 函数并不会立刻终止这个线程，该线程何时终止取决于操作系统的调度策略，因此，程序紧接着调用了 QThread 基类的 wait() 函数，它使得线程阻塞等待直到退出或超时。需要注意的是，terminate() 函数过于毒辣，它可能在线程执行的任意一步终止执行，从而产生不可预知的后果（如修改某个重要数据时），另外，它也没有留给线程任何清理现场的机会（如释放内存和锁等）。在这个实例中使用 terminate() 函数仅是演示方法，并不提倡使用。读者将发现当按下“停止”按钮后线程无法立刻停止，控制台的输出仍将持续一段时间，有时甚至长达数分钟，这正是 terminate() 函数无法立刻终止线程而 wait() 函数产生阻塞等待造成的。

要编译这个应用程序还必须在 .pro 文件中加入如下一行：

```
CONFIG += thread
```

它告诉 qmake 使用 Qt 库中的线程版本。

最后看一下运行结果，如表 10-1 所示。第一列为启动单一线程的运行结果，第二列为启动多个线程的运行结果。读者很容易看到，单一线程的输出是顺序打印的，而多线程的输出结果则是乱序打印的，这正是多线程应用程序的一大特点。



表 10-1 线程启动示例运行结果

单一线程运行结果	多线程运行结果
00000000
11111111	88888888
22222222	99999999
33333333	44444444
44444444	77777777
55555555	33333333
66666666	44444444
77777777	55555555
88888888	00000000
99999999	44444444
00000000	88888888
11111111	66666666
22222222	55555555
33333333	55555555
44444444	11111111
55555555	77777777
66666666	99999999
77777777	66666666
88888888	66666666
99999999	88888888
.....	99999999

总结起来多线程程序有以下特点：

- (1) 多线程程序的行为无法预期，当多次执行上述程序时，每一次的运行结果都不相同；
- (2) 线程的执行顺序无法保证，它与操作系统的调度策略和线程优先级等因素有关；
- (3) 线程的切换可能发生在任何时刻任何地点；
- (4) 线程对代码的敏感度高，代码的细微修改可能产生意想不到的结果。因此，为了有效使用线程，我们必须对其进行控制。

下面一节将介绍 Qt 线程同步互斥控制的基本方法。

10.2 线程互斥与同步

线程之间存在着相互制约的关系。这些关系可分为两类：

- (1) 互斥关系（亦称间接制约关系），即线程间因相互竞争使用独占型资源（互斥资源）所产生的制约关系。如线程间因争夺 I/O 设备而导致一方须等待另一方使用结束后方可使用。
- (2) 同步关系（亦称直接制约关系），是指为完成同一任务的伙伴线程间，因为需要在某些位置上协调它们的工作而相互等待、相互交互信息所产生的制约关系。

编写多线程程序面临的一个最具挑战性的问题就是如何让一个线程和另一个线程协同工作，避免出现竞争条件（race conditions）和数据破坏（data corruption）。为了使应用程序中的各个线程协同工作，线程需要根据应用需求在某些情况下进行必要的互斥与同步。

10.2.1 临界区问题

首先看一个例子，代码如下：

```
class Key
{
public:
    Key() { key = 0; }
    int creatKey() { ++key; return key; }
    int value() const { return key; }
private:
    int key;
};
```

类 Key 用来生成数据库表的主键值。该值从 0 开始递增并且不允许重复。在多线程环境下，这个类是不安全的，由于存在多个线程同时修改私有成员 key，其结果是不可预知的。虽然类 Key 产生主键的函数 creatKey() 只有一条语句执行修改成员变量 key 的值，但 C++ 的“++”操作符并不是原子操作，通常编译后它将被展开成如下三条机器指令：

- 将变量值载入寄存器
- 将寄存器中的值加 1
- 将寄存器中的值写回主存

假设当前的 key 值为 0，如果线程 A 和线程 B 同时将 0 值载入寄存器，执行加 1 操作并将加一后的值写回主存，结果是两个线程的执行结果将相互覆盖，实际上仅进行了一次加 1 操作，此时的 key 值为 1。如果线程 A 和线程 B 紧接着返回当前 key 值作为数据库表的主键值，两者将因获得一个重复的主键而产生错误。

为了保证类 key 在多线程环境下正确执行，上面的三条机器指令必须串行执行且不允许中途被打断（原子操作），即线程 A 在线程 B（或线程 B 在线程 A）之前完整执行上述三条机器指令。

实际上私有变量 key 是一个临界资源（CR，Critical Resource）。临界资源一次仅允许一个线程使用，它可以是一块内存、一个数据结构、一个文件，或者任何其他具有排他性使用的东西。程序中往往竞争使用临界资源，这些必须互斥执行的代码段称为“临界区（CS，Critical Section）”，该程序段实施对临界资源的操作。为了阻止问题产生，一次只能有一个线程进入临界区，通常有相关的机制或方法在程序中加入“进入”或“离开”临界区等操作，如果一个线程已经进入某个临界区，另一个线程就绝不可能在此时刻进入同一个临界区。

Qt 为实现线程的互斥与同步提供了以下几个常用类，它们是：QMutex、QMutexLocker、QReadWriteLocker、QReadLocker、QWriteLocker、QSemaphore 和 QWaitCondition。

10.2.2 使用 QMutex

QMutex 类提供一个保护一段临界区代码的方法，它每次只允许一个线程访问这段临界区代码。QMutex 类的 lock() 函数用来锁住互斥量，如果互斥量处于解锁状态，当前线程就会立即抓住并锁定它；否则当前线程就会被阻塞，直到持有这个互斥量的线程对它解锁。线程调用 lock() 函数后就会持有这个互斥量直到调用 unlock() 操作为止。QMutex 还提供了一个 tryLock() 函数，如果互斥量已被锁定，就立即返回。现在来看类 Key 使用 QMutex 进行互斥操作后的代码。

```

class Key
{
public:
    Key() { key = 0; }
    int creatKey() { mutex.lock(); ++key; return key; mutex.unlock(); }
    int value() const { mutex.lock(); return key; mutex.unlock(); }
private:
    int key;
    QMutex mutex;
};

```

现在新的问题又出现了，虽然在 `creatKey()` 函数中使用 `mutex` 进行了互斥操作，但是 `unlock()` 操作却不得不在 `return` 之后，从而导致 `unlock()` 操作永远无法执行。同样，`value()` 函数也存在这个问题。

Qt 提供的 `QMutexLocker` 类可以简化互斥量的处理，它在构造函数中接受一个 `QMutex` 对象作为参数并将其锁定，在析构函数中解锁这个互斥量。这样上面类 `Key` 中的问题终于有了一个完美的解决方法，代码如下：

```

class Key
{
public:
    Key() { key = 0; }
    int creatKey() { QMutexLocker locker(&mutex); ++key; return key; }
    int value() const { QMutexLocker locker(&mutex); return key; }
private:
    int key;
    QMutex mutex;
};

```

`locker` 作为局部变量会在函数退出时结束其作用域，从而自动对互斥量 `mutex` 解锁，这也正是想要的效果。实际应用中，一些互斥量锁定和解锁逻辑往往比复杂，并且容易产生错误，使用 `QMutexLocker` 类后，正如上面看到的，通常只需要一条语句。

10.2.3 使用 QSemaphore

Qt 中的信号量是由 `QSemaphore` 类提供的，信号量可以理解是对互斥量功能的扩展，互斥量只能锁定一次而信号量可以获取多次，它可以用来保护一定数量的同种资源。`acquire(n)` 函数用于获取 `n` 个资源，当没有足够的资源时调用者将被阻塞直到有足够的可用资源。`release(n)` 函数用于释放 `n` 个资源。`QSemaphore` 类还提供了一个 `tryAcquire(n)` 函数，在没有足够的资源时该函数会立即返回。信号量的典型用例是控制生产者和消费者之间共享的环形缓冲区，接下来看一个生产者 / 消费者的实例。

```

const int DataSize = 100000;
const int BufferSize = 8192;
int buffer[BufferSize];

```

生产者向 `buffer` 中写入数据，直到它到达终点，然后从起点重新开始覆盖已经存在的数据。消费者读取前者产生的数据，如图 10-2 所示。

生产者/消费者实例中对同步的需求有两处，如果生产者过快地生产数据，将会覆盖消费者还没有读取的数据；如果消费者过快地读取数据，将越过生产者并且读取到一些过期数据。

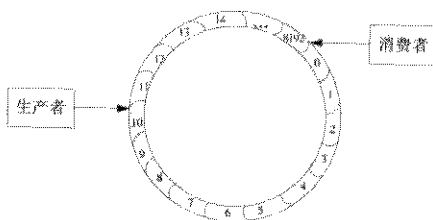


图 10-2 生产者/消费者模型

一种笨拙的方法是首先让生产者填满整个缓冲区，然后等待消费者读取整个缓冲区。另一种效率较高的方法是让生产者和消费者线程同时分别操作缓冲区的不同部分。为此使用了两个信号量：

```
QSemaphore freeBytes(BufferSize);
QSemaphore usedBytes(0);
```

`freeBytes` 信号量控制可被生产者填充的缓冲区部分。`usedBytes` 信号量控制可被消费者读取的缓冲区部分。这两个信号量是互为补充的，其中 `freeBytes` 信号量被初始化为 `BufferSize` (8192)，表示程序一开始有 `BufferSize` 个缓冲区单元可被填充，而信号量 `usedBytes` 被初始化为 0，表示程序一开始缓冲区中没有数据可供读取。

在这个示例中，每一个 `int` 字长被看作一个资源，实际应用中常会在更大的单位上进行操作，从而减小使用信号量带来的开销。接下来看生产者和消费者的具体实现代码。

```
class Producer : public QThread
{
public:
    void run();
};

void Producer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        freeBytes.acquire();
        buffer[i % BufferSize] = (i % BufferSize);
        usedBytes.release();
    }
}
```

生产者线程首先获取一个空闲单元，如果此时缓冲区被消费者尚未读取的数据填满，对 `freeBytes.acquire()` 函数的调用就会阻塞，直到消费者读取了这些数据为止。一旦生产者获取了某个空闲单元，就用当前的缓冲区单元序号填写这个缓冲区单元，然后调用 `usedBytes.release()` 把可用资源加 1，表示消费者此时可以读取这个刚刚填写的单元了。

```
class Consumer : public QThread
{
public:
    void run();
};
```

```

void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        usedBytes.acquire();
        fprintf(stderr, "%d ", buffer[i % BufferSize]);
        if(i % 16 == 0 && i != 0)
            fprintf(stderr, "\n");
        freeBytes.release();
    }
    fprintf(stderr, "\n");
}

```

消费者线程首先获取一个可被读取的单元, 如果缓冲区中没有包含任何可读取的数据, 对 `usedBytes.acquire()` 的调用将会阻塞, 直到生产者生产了一些数据。一旦消费者获取了这个单元, 会将这个单元的内容打印出来, 然后调用 `freeBytes.release()` 使这个单元变为空闲的, 以备生产者下次填充。

```

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    Producer producer;
    Consumer consumer;
    producer.start();
    consumer.start();
    producer.wait();
    consumer.wait();
    return 0;
}

```

`main()` 函数的功能比较简单, 负责启动生产者和消费者线程, 然后等待其各自执行完毕后自动退出。这个实例的运行结果如表 10-2 所示。

表 10-2 生产者/消费者实例运行结果

```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112
113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128
129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160
161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176
177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192
193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208
209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224
225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240
241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256

```

对生产者和消费者问题的另一个解决方法是使用 `QWaitCondition`，它允许线程在一定条件下唤醒其他线程。其中 `wakeOne()` 函数在条件满足时随机唤醒一个等待线程，而 `wakeAll()` 函数则在条件满足时唤醒所有等待线程。

10.2.4 使用 `QWaitCondition`

下面重写生产者和消费者实例，为了演示 `QWaitCondition` 的工作方式，这次使用一个生产者线程和两个消费者线程，并以 `QMutex` 作为等待条件。

```
const int DataSize = 10000;
const int BufferSize = 8192;
int buffer[BufferSize];

QWaitCondition bufferEmpty;
QWaitCondition bufferFull;
QMutex mutex;
int numUsedBytes = 0;
int rIndex=0;
```

在缓冲区之外声明了两个 `QWaitCondition` 变量：`bufferEmpty` 和 `bufferFull`，一个 `QMutex` 变量 `mutex`，一个表示存在多少“可用字节”的变量 `numUsedBytes`，以及用于指示当前所读取缓冲区位置的变量 `rIndex`。

在生产者中，首先检查缓冲区是否已填满，如果是则等待“缓冲区有空位”（`bufferEmpty` 变量）条件。当这个条件成立时，向缓冲区写入一个整数值，然后增加 `numUsedBytes` 变量，最后唤醒等待“缓冲区有可用数据”（`bufferFull` 变量）条件为“真”的线程。

```
void Producer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        mutex.lock();
        if (numUsedBytes == BufferSize)
            bufferEmpty.wait(&mutex);
        buffer[i % BufferSize] = numUsedBytes;
        ++numUsedBytes;
        bufferFull.wakeAll();
        mutex.unlock();
    }
}
```

for 循环中的所有语句需要使用互斥量加以保护，以保证其操作的原子性。最后对 `bool QWaitCondition::wait (QMutex * mutex, unsigned long time = ULONG_MAX)` 函数做一点说明。这个函数将互斥量解锁并在此等待，它带有两个参数，第一个参数为一个锁定的互斥量，第二个参数为等待时间。如果作为第一个参数的互斥量在调用时不是锁定的或出现递归锁定的情况，`wait()` 函数将立刻返回。调用 `wait()` 操作的线程使得作为参数的互斥量在调用前变为解锁定状态，然后自身被阻塞变为等待状态直到满足以下条件之一：

- 其他线程调用了 `wakeOne()` 或者 `wakeAll()` 函数，这种情况下将返回“true”值

- 第二个参数 `time` 超时 (以毫秒记), 该参数默认情况下为 `ULONG_MAX`, 表示永不超时, 这种情况下将返回 “false” 值

`wait()` 函数返回前会将互斥量参数重设置为锁定状态, 从而保证从锁定状态到等待状态的原子性转换。

```
void Consumer::run()
{
    forever {
        mutex.lock();
        if (numUsedBytes == 0)
            bufferFull.wait(&mutex);
        printf("%ul::[%d]=%d \n", currentThreadId (), rIndex, buffer[rIndex]);
        rIndex = (++rIndex)%BufferSize;
        --numUsedBytes;
        bufferEmpty.wakeAll();
        mutex.unlock();
    }
    printf("\n");
}
```

消费者线程的情况刚好相反, 它等待 “缓冲区有可用数据” (`bufferFull` 变量) 条件。当这个条件成立时, 打印当前线程号和 `rIndex` 变量指示的当前可读取数据, 然后将 `rIndex` 变量循环加 1, `numUsedBytes` 变量减 1, 最后唤醒等待 “缓冲区有空位” (`bufferEmpty` 变量) 条件的生产者线程。

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    Producer producer;
    Consumer consumerA;
    Consumer consumerB;
    producer.start();
    consumerA.start();
    consumerB.start();
    producer.wait();
    consumerA.wait();
    consumerB.wait();
    return 0;
}
```

主函数变化不大, 与前面例子不同的是这里启动了两个消费者线程 `consumerA` 和 `consumerB`。这两个线程读取相同的缓冲区, 为了不重复读取, 设置了全局变量 `rIndex` 用来表示当前读取缓冲区的位置, 并以互斥量保证其操作的原子性。为了区分究竟是哪一个消费者线程消耗了缓冲区里的数据, 在 `Consumer::run()` 函数中输出了当前线程的线程 ID, 方法是使用 `QThread` 类的 `currentThreadId ()` 静态函数。在 X11 环境下, 这个 ID 是一个 `unsigned long` 类型的值。

Qt 提供的另一个同步互斥类是 `QReadWriteLocker`, 它与 `QMutex` 十分相似, 但具有更高的效率。前者允许并发读取, 而后者不允许。具体来说, `QReadWriteLocker` 允许多个线程同时以只读方式访问, 而一旦有写入操作时, 所有线程将阻塞直到写入完成。因此, 在大量读取而偶尔写入的情况下

QReadWriteLocker 非常适合。对于上面的 waitconditon 实例, 如果使用 QReadWriteLocker 进行同步, 两个消费者线程可以实现并发读取, 但遗憾的是当前版本 Qt 4.3 的 QWaitCondition::wait() 只能接受 QMutex 类型的变量而无法以 QReadWriteLocker 类型的变量作为等待条件。

最后, Qt 为读写锁还提供了两个类, 它们分别是 QReadLocker 和 QWriteLocker。这两个类在构造函数中接受一个读写锁 (QReadWriteLocker), 从而可以很方便地在构造时对其进行读锁定或写锁定, 然后在析构时自动解除锁定。这一点与 QMutexLocker 和 Qmutex 的关系是一样的。

10.3 线程的其他问题

除了互斥与同步控制, 多线程编程还涉及死锁和优先级控制两个经典话题, 这一节将对此进行介绍, 希望大家能对多线程编程有一个较为全面的认识, 并在日后的工作实践中引起足够的重视, 但这并不是本书的重点, 有兴趣的读者可以查阅相关著作。Qt 还提供了线程本地存储的功能, 这将在本节中一起介绍。

10.3.1 优先级问题

实际应用中, 通常希望根据各线程所运行任务性质的不同让某些线程优先执行, 这就需要对各线程进行必要的优先级设置。具有较高优先级的线程通常会获得更多的调度机会。

Qt 在线程基类 QThread 中引入了线程优先级的概念, 并提供了 Priority priority() const 函数和 void setPriority(Priority priority) 用于获取和设置当前正在运行线程的优先级。另一种设置线程优先级的方法是在启动线程的 void start(Priority priority = InheritPriority) 函数中, 它带有一个优先级参数, 前面的应用均使用了它的默认值。对于优先级类型 Priority, QThread 类中定义它为一个公有的枚举类型, 如下所示:

```
class Q_CORE_EXPORT QThread : public QObject
{
.....
public:
    enum Priority {
        IdlePriority,
        LowestPriority,
        LowPriority,
        NormalPriority,
        HighPriority,
        HighestPriority,
        TimeCriticalPriority,
        InheritPriority
    };
.....
}
```

其中每一项枚举值的含义如表 10-3 所示。

介绍到这里, Qt 线程优先级的问题似乎很清楚了, 只要选择一种线程优先级类别, 然后使用 QThread 中的 start() 或 setPriority() 函数进行设置就可以了, 所有工作不过是一行代码的事情。但实际情况并非如此, 由于 Qt 是一种跨平台的开发工具, 它所作的线程抽象最终需要映射到实际操作系统平



台的线程实现上。在 Linux 操作系统中,线程实现遵循 Posix 标准,这种实现将 Qt 所做的线程优先级抽象化为乌有。为了说明问题,必须首先了解一下 Posix 的线程调度问题。

表 10-3 Qt 线程优先级类别

标 识	值	描 述
QThread::IdlePriority	0	在没有其他线程运行时调度
QThread::LowestPriority	1	比 LowPriority 获得的调度机会少
QThread::LowPriority	2	比 NormalPriority 获得的调度机会少
QThread::NormalPriority	3	操作系统默认调度优先级
QThread::HighPriority	4	比 NormalPriority 获得的调度机会多
QThread::HighestPriority	5	比 HighPriority 获得的调度机会多
QThread::TimeCriticalPriority	6	尽可能调度
QThread::InheritPriority	7	继承创建线程的优先级,Qt 默认行为

Linux 中遵循 Posix 标准的线程是按不同的策略调度的,这些策略如表 10-4 所示。

表 10-4 Linux 中的线程调度策略

策 略	类 型	优 先 级
SCHED_FIFO	先进先出	1~99
SCHED_RR	轮转	1~99
SCHED_OTHER	其他	0

较大的优先级值对应较高的优先级,具有相同优先级的线程根据它们的调度策略来竞争处理器资源。SCHED_FIFO 调度策略为一个特定的优先级中处于可运行状态的线程使用了一个队列,转成可运行状态的阻塞线程被放入与它们的优先级相对应的队列末尾,而被抢占的运行线程则放在队列的前面;在 SCHED_RR 调度策略中,当运行线程耗尽了它的时间片之后,就被放入它的优先级队列的末尾,除此之外,它的行为与 SCHED_FIFO 调度策略类似;SCHED_OTHER 调度策略通常由操作系统决定,其中最为常见的是抢占式调度策略。SCHED_RR 与 SCHED_FIFO 调度策略具有 99 个优先级级别,但它们只能在 root 用户下使用;SCHED_OTHER 调度策略是 Linux 系统的默认调度策略,可以被所有用户使用,但它只有一个优先级级别 0。现在问题呈现出来了,Qt 没有提供设置线程调度策略的接口,那么它所定义的线程优先级在 Linux 系统中究竟映射成了什么?开源的 Qt 源代码提供了线索,答案在“/src/corelib/thread”目录下的 qthread_unix.cpp 文件中。这个文件涉及线程优先级映射的有两处,分别在函数 start()和函数 setPriority()中,但内容几乎相同,现将它摘录下来进行分析。

```
switch (priority) {
    case InheritPriority:
    {
        pthread_attr_setinheritsched(&attr, PTHREAD_INHERIT_SCHED);
        break;
    }
    default:
    {
        int sched_policy;
        if (pthread_attr_getschedpolicy(&attr, &sched_policy) != 0) {
            // failed to get the scheduling policy, don't bother

```



```

// setting the priority
qWarning("QThread::start: Cannot determine default scheduler policy");
break;
}

int prio_min = sched_get_priority_min(sched_policy);
int prio_max = sched_get_priority_max(sched_policy);
if (prio_min == -1 || prio_max == -1)
{
    // failed to get the scheduling parameters, don't
    // bother setting the priority
    qWarning("QThread::start: Cannot determine scheduler priority
        range");
    break;
}

int prio;
switch (priority) {
case IdlePriority:
    prio = prio_min;
    break;
case TimeCriticalPriority:
    prio = prio_max;
    break;
default:
    // crudely scale our priority enum values to the prio_min/prio_max
    prio = (((prio_max - prio_min) / TimeCriticalPriority) * priority)
        + prio_min;
    prio = qMax(prio_min, qMin(prio_max, prio));
    break;
}

sched_param sp;
sp.sched_priority = prio;
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedparam(&attr, &sp);
break;
}
}

```

这段代码首先判断优先级是否为 `InheritPriority`，如果是就将它映射到 Posix 的 `PTHREAD_INHERIT_SCHED` 值，表示使用创建线程的线程设置，然后调用的 `pthread_attr_setinheritsched(&attr, PTHREAD_INHERIT_SCHED)` 函数完成设置并返回。否则需要先使用 `pthread_attr_getschedpolicy(&attr, &sched_policy)` 函数获取当前的调度策略 `sched_policy`，然后分别使用函数 `sched_get_priority_min(sched_policy)` 和 `sched_get_priority_max(sched_policy)` 获取这个调度策略下优先级的最小值 `prio_min` 与最大值 `prio_max`。接下来是将 Qt 的优先级级别分别映射到 `[prio_min, prio_max]` 间，映射关系如表 10-5 所示。

最后需要生成一个调度策略参数 `sched_param sp`，将映射后的线程优先级置入这个参数 `sp.sched_priority = prio`，然后调用 `pthread_attr_setschedparam(&attr, &sp)` 函数完成最终设置。分析完上述代码，大家可能比较失望，因为其中并没有出现任何设置调度策略的地方，它使用了 Linux 的默认



调度策略 `SCHED_OTHER`。因此，Qt 所定义的全部线程优先级在 Linux 下均被映射成了“0”。此外，这段代码中还出现了大量的 Posix 线程函数，限于篇幅这里不做逐一解释，它们在 Linux 的文档中有详细注释，有兴趣的读者可以查阅。

表 10-5 Qt 优先级映射关系

Qt 优先级	Linux 优先级
<code>IdlePriority</code>	<code>prio_min</code>
<code>TimeCriticalPriority</code>	<code>prio_max</code>
<code>LowestPriority</code> , <code>LowPriority</code> , <code>NormalPriority</code> , <code>HighPriority</code> , <code>HighestPriority</code>	$\text{prio} = (((\text{prio_max} - \text{prio_min})$ $/ \text{TimeCriticalPriority}) * \text{priority})$ $+ \text{prio_min}$ $\text{prio} = \text{qMax}(\text{prio_min}, \text{qMin}(\text{prio_max},$ $\text{prio}))$

10.3.2 死锁及优先级反转问题

之所以将死锁和优先级反转问题一起讨论，是因为两者在很大程度上都是因为线程的互斥与同步操作不当引起的。首先看一下死锁问题。

死锁是操作系统中的一个经典话题，对它的研究已经比较透彻，在此不做讨论，这里只引用结论。死锁的产生必须具备以下 4 个必要条件。

1. 互斥条件

指进程对所分配到的资源进行排他性使用，即在一段时间内某资源只由一个进程占有。如果此时有其他进程请求该资源，申请者只能阻塞，直到占有该资源的进程用毕释放。

2. 请求保护条件

进程已经保持了至少一个资源，但又提出了新的资源申请，而该资源已被其他进程占有，此时请求进程阻塞，但对已获得的其他资源保持不放。

3. 不可剥夺条件

进程已获得的资源，在未使用完全之前不能被剥夺，只能在使用完时由自己释放。

4. 环路等待条件

在发生死锁时，必然存在一个“进程/资源”的环形链。

上述结论虽然是操作系统中的定义，但对于造成线程的死锁同样适用。其中，在线程同步中经常使用的 `QMutex` 就是一种互斥资源，多个线程对它的滥用，如忘记解锁或程序异常时造成的锁无法释放，都有可能促成死锁的 4 个必要条件。实际上，使用了互斥与同步机制的程序中都有潜在出现死锁的可能。目前用于处理死锁的方法有以下 4 种：

1. 死锁预防

通过设置某些限制条件破坏产生死锁的 4 个必要条件中的一个或几个，来防止发生死锁。

2. 死锁避免

不需要事先采取限制措施破坏产生死锁的必要条件，而是在资源的动态分配过程中，用某种方法

防止系统进入不安全状态，从而避免发生死锁。

3. 死锁检测

系统在运行过程中发生死锁。但可通过系统设置的检测机构及时地检测出死锁的发生，然后采取措施，从系统中将已发生的死锁清除掉。

4. 死锁解除

这是与死锁检测相配套的一种措施，用于将进程从死锁状态下解脱出来。

这4种方法在操作系统中均有应用，但在处理因为线程造成的应用程序死锁问题上尚未看到很好的应用。Posix基本标准的各种实现可能检测不到这些死锁，它的立场是，不要求基本标准的实现牺牲效率来保护程序员免受自己不良编程带来的影响。但Posix的几种扩展都允许进行更广泛的错误检查和死锁检测。Qt在死锁检测方面也做了一些工作，当程序可能出现死锁时Qt将报告错误，如因互斥量引起的死锁错误报告形式为“QMutex::lock:Deadlock detected in thread 30836479201”。但是Qt在文档中并没有提及这部分内容，当前开发人员还是只能依靠小心谨慎来消除死锁隐患。

当带有优先级的线程和同步互斥机制混在一起时，就可能产生另一个问题——“优先级反转”。火星探路者任务就是一个很著名的例子，下面介绍这个例子，从中可以看到产生优先级反转的原因。

探路者在1997年7月4日进行了完美的火星着陆，然后开始收集并向地球传回大量的科学数据。着陆几天之后，宇宙飞船开始进行完全的系统复位，但每次系统复位都要使数据收集延迟一天。后来人们发现，火星探路者的问题是一个互斥量引起的优先级反转问题。一个负责收集气象数据的线程以较低的优先级周期性运行，这个线程要获取数据总线的互斥量来发布它的数据。一个周期性的高优先级线程也要获取这个互斥量，这个线程偶尔也会阻塞，以等待低优先级线程释放互斥量。这些线程只是在很短的时间内需要互斥量，因此表面上看起来不会有什么问题。但是当低优先级线程持有互斥量时，一个长时间运行的中等优先级的通信线程偶尔会抢占低优先级线程，这就会使高优先级线程延迟很长时间运行。

死锁和优先级反转问题往往具有很强的隐蔽性和偶发性，错误复现概率低，因此发现和调试都很困难。目前，程序员大多只能凭借自己的经验来应对。笔者在此提出这两个问题，也是希望大家在使用Qt进行多线程编程时引起足够的重视。

10.3.3 本地存储问题

10.2节中的所有实例，线程在访问全局变量时都使用了各自不同的互斥方法。但是某些多线程应用程序却需要在不同的线程中保存全局变量不同的值。这种变量通常被称为线程本地存储(thread-local storage, TLS)或线程特定数据(thread-specific data, TSD)。最典型的应用是全局错误状态变量，为了保证在一个线程中的修改不会影响其他线程，每一个线程需要拥有这个全局变量的一个副本。

Qt提供了QThreadStorage类用于存储线程的单独数据。这个类是一个模板类，由于编译器的限制，QThreadStorage类目前只能存储指针。其中setLocalData()函数为调用线程存储一份独立的数据，这个数据随后可以由localData()函数访问。QThreadStorage类持有存储数据所有者线程的信息（存储的数据必须以new操作符在堆空间里分配），当所有者线程正常或非正常退出时，所存储的数据将自动释放。hasLocalData()函数允许程序员判断线程中是否有经setLocalData()函数存储的数据，这个函数在初始化的过程中十分有用。下面看一个示例。

```
QThreadStorage<QCache<QString, SomeClass>*> caches;
```



```
void cacheObject(const QString &key, SomeClass *object)
{
    if (!caches.hasLocalData())
        caches.setLocalData(new QCache<QString, SomeClass>);

    caches.localData()->insert(key, object);
}

void removeFromCache(const QString &key)
{
    if (!caches.hasLocalData())
        return;
    caches.localData()->remove(key);
}
```

对于每一个调用 `cacheObject()` 函数的线程, `QThreadStorage` 类将为它存储一份独立的数据, 而在调用 `removeFromCache()` 函数时将其删除, 当调用者线程退出时, `QThreadStorage` 类实例 `caches` 将自动被释放。

10.4 Qt 的线程机制

在具有图形用户界面的 Qt 应用程序中, 主线程由 GUI 线程充当, 该线程是 Qt 中唯一可以执行 GUI 相关操作的线程, 即一个具有图形用户界面的 Qt 应用程序只能有一个 GUI 线程。但是, 它可以同时拥有一个或多个非 GUI 线程作为工作线程处理其他耗时操作。这样处理之后, 即使在负载很重的情况下, 应用程序也可以保证图形用户界面的响应。

编写 Qt 多线程应用程序的一个棘手问题是, 如何在 GUI 线程和非 GUI 线程间进行通信。上一节介绍的互斥量、信号和等待条件等均无法满足与 GUI 线程通信的需求, 因为它们会锁住事件循环造成用户界面冻结。为了有效地与 GUI 线程通信, Qt 提供了一整套机制, 但也对程序的编写做出了许多限制。为了更好地理解这些内容, 首先需要将必需的基本概念阐述清楚, 这可能是一个枯燥的过程, 首次阅读时可以跳过这部分内容而直接进入后面的示例部分, 在产生疑惑时再回头寻求答案。

10.4.1 可重入与线程安全

在 Qt 文档中, 可重入 (reentrant) 和线程安全 (thread-safe) 均用于描述能用在多线程环境下的函数, 它们的定义分别如下:

- 如果一个函数能同时被多个线程调用, 并且为每一个调用者提供一份单独的数据, 那么称这个函数是可重入的。
- 如果一个函数能同时被多个线程调用, 并且所有的调用者引用同一份数据, 访问数据时串行处理, 那么称这个函数是线程安全的。

下面为上述概念进行一下扩展, 如果一个类中的所有函数在这个类的各个实例中可以被多个线程同时调用, 那么称这个类是可重入的; 同样, 如果类的一个实例可以同时被多个线程操作, 那么称这个类是线程安全的。这里使用的术语并非标准定义, Posix 对此在 C 接口上就有不同的描述。但对于像 Qt 这种面向对象的 C++ 库, 必须采用上述定义。

事实上, 许多 C++ 类由于只引用自身的成员变量, 因此天然就是可重入的。如 10.2.1 节的类 `Key`

就是可重入的,但这个类并不是线程安全的。通常情况下,任何没有被全局引用或被其他共享对象引用的 C++ 对象都是可重入的。Qt 的文档中对其所提供的类在这两点上均有详细注释。例如在 QObject 类的注释中会发现如下注意事项:

Note: All the functions in this class are reentrant, except connect(), connect(), disconnect(), and disconnect().

对于 Qt 类是否是可重入的以及是否是线程安全的可做如下概括:

QObject 类是可重入的。大多数非 GUI 类如 QTimer、QTcpSocket、QUdpSocket、QHttp、QFtp、和 QProcess 也是可重入的,从而使得这些类可以同时用于多个线程。但是 GUI 类,如 QWidget 及其子类均是不可重入的,它们只能用于主线程。Qt 中线程安全的类包括 QThread、QMutex、QMutexLocker、QReadWriteLocker、QReadLocker、QWriteLocker、QSemaphore、QWaitCondition 和 QThreadStorage。此外,函数 QApplication::postEvent()、QApplication::removePostedEvent()和 QApplication::wakeUp()也是线程安全的。但是 QObject 及其子类以及整个事件分发机制都不是线程安全的。在介绍线程的事件循环机制时,还将进一步指出在多线程环境下使用 Qt 类的种种约束。

10.4.2 线程与事件循环

GUI 线程是唯一允许创建 QApplication 对象并且对它调用 exec()函数的线程。在调用了 exec()函数后,GUI 线程要么等待一个事件,要么处理一个事件。每一个线程都可以有自己的事件循环,如图 10-3 所示。起始线程通过 QCoreApplication::exec()函数启动事件循环,其他非 GUI 线程通过 QThread::exec()启动各自的事件循环。与 QCoreApplication 类似,QThread 也提供了退出函数 exit()和 quit()槽。

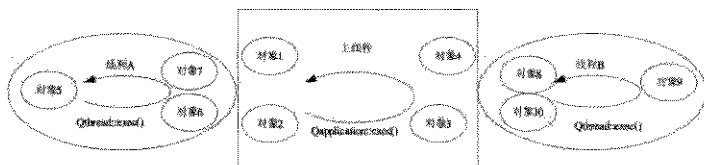


图 10-3 Qt 线程事件模型

线程中的事件循环机制使得在线程中使用某些需要事件机制的非 GUI 类成为可能,例如 QTimer 类、QTcpSocket 类和 QProcess 类等。它同样使处于不同线程内的对象可以进行信号和槽的连接。

一个线程中所创建 QObject 对象(这里的 QObject 对象指所有 QObject 以及从它派生的 Qt 子对象,下同)在这个线程中被称为是活跃的(live)(以下称这个 QObject 对象属于某个线程)。发往这个 QObject 对象的事件由其创建线程的事件循环派发。QObject 对象还可以通过 QObject::thread()方法获取其所属的线程。需要注意的是,在 QApplication 对象创建之前构造的 QObject 对象在调用 QObject::thread()函数时将返回“0”。这意味着主线程只处理这些 QObject 对象的 posted 事件,而不处理其他任何事件,原因是这些 QObject 对象不属于任何线程。但是可以通过 QObject::moveToThread()函数改变 QObject 对象所属的线程。需要注意的是,如果这个 QObject 对象有父对象将无法移动。

任何时候手动调用线程安全的 QCoreApplication::postEvent()函数向任意线程创建的对象发消息,这个消息最终都会被派发到创建这个对象所属线程的事件循环。事件过滤器在多个线程中同样被支持,唯一的限制是监控对象和被监控对象必须同属一个线程。类似的,QCoreApplication::sendEvent()函数也只能向调用线程中的对象发送消息。



如果无法保证一个 `QObject` 对象当前未处理事件, 那么删除或通过其他方式访问其他线程的 `QObject` 对象是不安全的。删除属于其他线程的 `QObject` 对象的安全办法是使用 `QObject::deleteLater()` 函数, 这个函数会发出一个 `DeferredDelete` 事件, 并最终被目标 `QObject` 对象所属线程的事件循环接收处理, 达到删除该对象的目的。

如果所启动的线程中没有事件循环, 那么事件将不会发往该线程所创建的 `QObject` 对象。例如, 如果某个线程启动了一个 `QTimer` 实例而未调用 `exec()` 操作, 那么 `QTimer` 将永远不会发出 `timeout()` 信号。同样, 调用 `deleteLater()` 函数也将不起作用。这个限制对于主线程同样适用。

需要时刻留意的是, 在访问其他线程所创建的 `Qt` 对象的时候, 事件循环机制随时都可能将事件分发到当前正在处理的 `QObject` 对象。如果在当前线程下调用了属于其他线程的 `QObject` 对象中的某个函数, 并且这个对象随时可能接收事件, 那么在访问这个对象内部数据时必须进行互斥操作, 否则可能产生数据崩溃或其他不可预期的后果。`QObject` 对象还必须遵循如下约束:

- `QObject` 子对象必须在其父对象所属的线程中创建。这意味着绝不要将 `QThread` 对象 (this 指针) 作为父对象创建其他对象, 因为 `QThread` 对象自身就是在另一个线程中创建的。
- 事件驱动类只能用于单线程。最典型的这类应用是时间机制类和网络模块类。例如, 不能在对象所不属于的线程中启动一个定时器或者连接套接字。
- 必须保证在某个线程中创建的所有对象在该线程释放前首先被释放。

和其他对象一样, `QThread` 对象自构造时在其调用线程中就是活跃的, 而不是在调用了 `QThread::run()` 函数才成为活跃的。因此, 在 `QThread` 子类中提供槽函数是不安全的, 除非对所有的变量施加了互斥操作。另一方面, 仍然可以在 `QThread::run()` 函数中安全地发出信号, 因为信号发送函数是线程安全的。

最后一点需要说明的是, `Qt` 使用了一种称为隐式数据共享的技术来优化其值类 (value class), 最典型的是 `QString` 类和 `QImage` 类 (这项技术参见第 13 章)。但在一般人的印象中, 由于隐式数据共享经常需要进行引用计数, 因此与多线程是无法兼容的。一种解决办法是对引用计数进行同步, 但这将大大降低访问速度。之前的 `Qt` 版本没有提供满意的解决方案, 但从 `Qt 4` 开始, 使用了隐式共享技术的类可以和其他类一样在线程间拷贝, 这些类完全是可重入的。实际上, `Qt 4` 使用了原子引用计数操作, 这项技术速度非常快, 它根据不同的平台由汇编语言实现。

10.4.3 线程与信号/槽机制

`QThread` 继承 `QObject`, 它可以发出信号表示当前线程开始执行或退出, 并且还提供了其他一些槽函数。十分重要的一点是 `QObject` 对象可以用于多线程, 其发出的信号能够激活其他线程的槽函数, 并且还可以向属于其他线程的对象发送事件。这些都是线程事件循环机制的功劳。从 `Qt 4` 开始信号和槽机制被扩展为可以支持跨线程的连接, 这大大方便了 GUI 线程和其他工作线程间的通信。目前, `Qt` 支持以下三种信号和槽的连接方式:

- **直接连接方式** (direct connections), 这种方式下信号发出后相应的槽函数将立即被调用。这个槽函数在发出信号的线程中执行, 而不一定必须在接收对象所属的线程中。
- **排队连接方式** (queued connections), 这种方式下信号发出后需等到接收对象所属线程的事件循环取得控制权时才调用响应的槽函数。这个槽函数在信号接收对象所属的同一个线程中执行。
- **自动连接方式** (auto connections), 这种方式是 `Qt` 的默认连接方式, 如果信号的发出与接收这个信号的对象同属一个线程, 那么工作方式与直接连接方式相同, 否则与排队连接方式相同。

所选择的连接类型可以通过参数传递给 `connect()` 函数。需要注意的是, 当信号的发送者和接收者分属不同的线程且接收者线程存在自身的事件循环时, 使用直接连接方式是不安全的。

在实际应用中, 试图在非主线程中调用 GUI 类是不可能的, 通常的解决方法是将那些耗时的操作放在独立的工作线程中, 当这些工作线程得出结果后再将结果放到主线程中显示。跨线程的信号和槽机制打开了工作线程和 GUI 线程通信的方便之门, 最后来看一个多线程的网络示例。

10.4.4 多线程网络示例

这个实例实现了一个多线程的网络时间服务器, 每当有客户请求到达时, 这个服务器将启动一个新线程为它返回当前的时间, 服务完毕后这个线程将自动退出, 同时用户界面会显示当前已接受请求的次数。程序运行的效果如图 10-4 所示。



图 10-4 多线程时间服务器实例

```
class TimeServer : public QTcpServer
{
    Q_OBJECT

public:
    TimeServer(QObject *parent = 0);
protected:
    void incomingConnection(int socketDescriptor);
private:
    Dialog *dlg;
};
```

首先需要实现一个 TCP 服务端类 `TimeServer`, 这里直接从 `QTcpServer` 类继承, 并重写了其虚函数 `void incomingConnection(int socketDescriptor)`。这个函数在 TCP 服务端有新的连接时被调用, 参数为所接受新连接的套接字描述符。私有成员 `Dialog *dlg` 用于记录创建这个 TCP 服务端对象的父类, 这里是界面指针, 借助这个指针, 将线程发出的消息关联到界面的槽函数中。

```
TimeServer::TimeServer(QObject *parent)
    : QTcpServer(parent)
{
    dlg = (Dialog*)parent;
}
```

构造函数十分简单, 这里用传入的父类指针 `parent` 初始化私有变量 `dlg` 就可以了。

```
void TimeServer::incomingConnection(int socketDescriptor)
{
    TimeThread *thread = new TimeThread(socketDescriptor, 0);
```



```

connect(thread, SIGNAL(finished()), dlg, SLOT(showResult()),
        Qt::QueuedConnection);
connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()),
        Qt::DirectConnection);
thread->start();
}

```

在重写的虚函数 `incomingConnection()` 中, 首先以返回的套接字描述符 `socketDescriptor` 创建一个工作线程 `TimeThread`, 然后将这个线程的结束消息 `finished()` 分别关联到界面显示类的槽函数 `showResult()` 用于显示请求计数, 以及线程自身的槽函数 `deleteLater()` 用于结束线程。一切准备工作完成后启动这个线程。需要注意的是, 在第一个 `connect` 操作中, 使用了排队连接方式, 第二个 `connect` 操作中使用了直接连接方式, 原因在于前一个信号是跨线程的, 后一个信号是在同一个线程中, 当然也可以省略 `connect()` 函数的最后一个参数, 而采用 Qt 的自动连接选择方式。另一个需要注意的是, 由于工作线程中存在网络事件, 因此不能被外界线程销毁, 这里使用了延迟销毁函数 `deleteLater()` 保证由工作线程自身销毁, 这一点在 10.4.2 节中已做说明。

```

class TimeThread : public QThread
{
    Q_OBJECT

public:
    TimeThread(int socketDescriptor, QObject *parent);
    void run();
signals:
    void error(QTcpSocket::SocketError socketError);
private:
    int socketDescriptor;
};

```

工作线程 `TimeThread` 由 `QThread` 类继承而来, 这里将重写重要的虚函数 `run()`。此外, 还定义了一个出错信号 `void error(QTcpSocket::SocketError socketError)` 和一个私有的套接字描述符 `socketDescriptor`。

```

TimeThread::TimeThread(int socketDescriptor, QObject *parent)
    : QThread(parent), socketDescriptor(socketDescriptor)
{
}

```

构造函数十分简单, 这里仅是初始化了私有套接字描述符。

```

void TimeThread::run()
{
    QTcpSocket tcpSocket;
    if (!tcpSocket.setSocketDescriptor(socketDescriptor)) {
        emit error(tcpSocket.error());
        return;
    }
    QDateTime time;
    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_3);
}

```



```

uint time2u = QDateTime::currentDateTime().toTime_t();
out << time2u;
tcpSocket.write(block);
tcpSocket.disconnectFromHost();
tcpSocket.waitForDisconnected();
}

```

虚函数 `run()` 是工作线程的实质所在，当在 `TimeServer::incomingConnection()` 函数中调用了 `start()` 函数后，这个虚函数开始执行。它首先创建一个 `QTcpSocket` 类并置以从构造函数中传入的套接字描述符，用来向客户端传回服务器端的当前时间。如果出错，发出 `error(tcpSocket.error())` 信号报告错误；否则，开始获取当前时间并将它传回客户端，然后断开连接等待返回。使用 `QTcpSocket` 类传数据的知识在第9章网络中已有详细介绍，不再赘述。这里着重介绍一下时间数据的传输格式，Qt 虽然可以很方便地通过 `QDateTime` 类的静态函数 `currentDateTime()` 获取一个时间对象，但类结构是无法直接在网络间传输的，此时需要将它转换成一个标准的数据类型后再传输。幸运的是 `QDateTime` 类提供了 `uint toTime_t()` `const` 函数，这个函数返回当前自 1970-01-01 00:00:00 经过了多少秒，为一个 `uint` 类型，可以将这个值传输给客户端。在客户端方面，使用 `QDateTime` 类的 `void setTime_t (uint seconds)` 将这个时间还原。

```

class Dialog : public QDialog
{
    Q_OBJECT

public:
    Dialog(QWidget *parent = 0);
public slots:
    void showResult();
private:
    QLabel *statusLabel;
    QLabel *reqStatusLabel;
    QPushButton *quitButton;
    TimeServer *server;
    int count;
};

```

界面类 `Dialog` 比较简单，它实际上就是一个对话框。在此定义了一个用于显示请求次数的槽函数 `void showResult()`，以及用于显示监听端口的标签 `statusLabel`，用于显示请求次数的标签 `reqStatusLabel`，退出按钮 `quitButton`，TCP 服务端 `server` 和请求次数计数器 `count`。

```

Dialog::Dialog(QWidget *parent)
    : QDialog(parent), count(0)
{
    server = new TimeServer(this);
    statusLabel = new QLabel;
    reqStatusLabel = new QLabel;
    quitButton = new QPushButton(tr("退出"));
    quitButton->setAutoDefault(false);
    if (!server->listen()) {
        QMessageBox::critical(this, tr("多线程时间服务器"),
            tr("无法启动服务器: %1.",
                .arg(server->errorString()));
    }
}

```



```
close();  
return;  
}  
statusLabel->setText(tr("时间服务器运行在端口: %1.\n")  
                    .arg(server->serverPort()));  
connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));  
QHBoxLayout *buttonLayout = new QHBoxLayout;  
buttonLayout->addStretch(1);  
buttonLayout->addWidget(quitButton);  
buttonLayout->addStretch(1);  
QVBoxLayout *mainLayout = new QVBoxLayout;  
mainLayout->addWidget(statusLabel);  
mainLayout->addWidget(reqStatusLabel);  
mainLayout->addLayout(buttonLayout);  
setLayout(mainLayout);  
setWindowTitle(tr("多线程时间服务器"));
```

构造函数 `Dialog` 完成了两件事情：一件是初始化界面，另一件是启动服务器端的网络监听。这两方面的知识在前面已有详细介绍，这里就不再逐一解释了。

```
void Dialog::showResult()  
{  
    reqStatusLabel->setText(tr("第%1次请求完毕.\n").arg(++count));  
}
```

槽函数 `showResult()` 功能十分简单，它在标签 `reqStatusLabel` 上显示当前的请求次数，并将请求计数 `count` 加 1。但是有一个问题，由于这个槽函数是被多个线程激活的，其请求计数 `count` 是否需要使用互斥量、信号量等保护手段？实际上这种担心是没有必要的，槽函数 `showResult()` 虽然被多个线程激活，但调用入口只有一个，那就是主线程的事件循环。多个线程的激活信号最终会在主线程的事件循环中排队调用槽函数 `showResult()`，从而保证了 `count` 变量的互斥访问，因此 `showResult()` 函数是一个天然的临界区。

10.5 小 结

Qt 中的多线程编程主要涉及两方面内容：一方面是传统的线程操作，另一方面是与 Qt 事件机制相关的操作。这一章介绍了线程的启停、同步等基本操作，以及一些线程的相关知识。在此基础之上，还详细介绍了 Qt 的线程机制，以及它和事件机制的关系，最终较为全面地向大家展示了 Qt 多线程编程的各个方法。

多线程编程是一个大而复杂的话题，中间有许多比较专业的概念和知识，在这一章中仅仅介绍了必备的基础知识以备读者理解本章内容。重点还是放在如何使用 Qt 开发多线程程序，而不是多线程主题自身。后者有很多书籍对此进行了专题讨论，有兴趣的读者可以参考一些资料：

Threads Primer: A Guide to Multithreaded Programming

Thread Time: The Multithreaded Programming Guide

Pthreads Programming: A POSIX Standard for Better Multiprocessing

《Win32 多线程程序设计》

第 11 章 事件处理

事件由窗口系统或 Qt 自身产生,用以响应各种行为或情况。如当按下或释放鼠标键时,会产生鼠标事件;当某个窗口第一次显示时,会产生一个 `paintEvent` 事件用来告知窗口绘制自身,从而使得该窗口可见。大多数事件是作为用户行为的响应而产生的,但是也有例外,如定时器、网络数据之类的事件则是由系统单独产生的。

11.1 事件机制

Qt 中定义的事件是一个从 `QEvent` 类继承而来的对象,它表示应用程序内部或外部发生了某些应用程序自身必须知道的事情。任何从 `QObject` 类派生的对象均可以通过 `QObject::event()` 方法接收事件。事件产生时,Qt 会创建一个合适的 `QEvent` 对象或其子对象,然后通过调用 `QObject` 类的 `event()` 函数将这个事件对象传给特定的 `QObject` 对象或其子对象。需要指出的是,`event()` 函数自身并不处理事件,而是根据事件类型调用相应的事件处理器,其返回值告知这个事件是否被接受并进行了处理。例如,`QWidget` 类中的 `event()` 函数实现将鼠标、键盘和重绘等常见事件交给 `mousePressEvent()`、`keyPressEvent()` 和 `paintEvent()` 这些特定的事件处理器进行处理。

在 Qt 内部,Qt 通过由函数 `QCoreApplication::exec()` 启动的主事件循环不停抓取事件队列中的本地窗口事件,然后将它们转换成对应的 `QEvent` 对象,并最终将这些 `QEvent` 对象发送到目的 `QObject` 对象来完成上述一系列事件的处理动作。

需要注意的是,不应该混淆事件和信号这两个概念。通常,在使用已有的窗口部件时,信号是十分有用的,而在自定义新的窗口部件时,事件则更为重要。举例来讲,当使用 `QPushButton` 类时,程序员往往只关注其 `clicked()` 信号,而很少关心底层的鼠标或者键盘事件。但当自定义一个类似 `QPushButton` 的类时,程序员就不得不编写一定量的代码来处理鼠标或者键盘事件,并在适当的时候发送 `clicked()` 信号。从这个角度讲,事件比信号更底层。

11.1.1 事件来源与类型

通常事件来源多种多样,最常见的有来自窗口系统的 `QMouseEvent` 和 `QKeyEvent` 事件,以及来自系统的 `QTimerEvent` 事件,还有一些事件则来自应用程序。

Qt 为多数事件定义了特定的类,值得关注的有 `QResizeEvent`、`QPaintEvent`、`QMouseEvent`、`QKeyEvent` 和 `QCloseEvent` 类。每一个类均从 `QEvent` 继承而来,并加入了本事件特定的功能函数。例如,`QResizeEvent` 事件类加入了 `size()` 和 `oldsize()` 函数,用来发现窗口部件尺寸如何变化。

Qt 中的所有事件类型由 `QEvent::Type` 枚举定义,这些类型可以作为信息源动态找出具体的事件子对象,例如:

```
bool MyWidget::event(QEvent *event)
```



```
{  
    if (event->type() == QEvent::KeyPress)  
    {  
        QKeyEvent *keyEvent = static_cast<QKeyEvent*>(event);  
        ....  
        return true;  
    }  
    return QWidget::event(event);  
}
```

仔细观察文档中 `QEvent::Type` 枚举还可以发现, 一些事件类支持多种事件类型, 如 `QMouseEvent` 支持鼠标单击、双击、移动, 以及其他相关操作。

11.1.2 事件处理方法

Qt 提供了处理事件的 5 种方式。

1. 重新实现特定的事件处理器

重新实现如 `mousePressEvent()`、`keyPressEvent()` 和 `paintEvent()` 这些 Qt 事先已定义的事件处理器是进行事件处理的最简单方式。

2. 重新实现 `QObject::event()` 函数

通过重新实现 `event()` 函数, 可以在事件到达特定的事件处理器之前截获并处理它们。这种方法可以用来覆盖已定义事件的默认处理方式, 也可以用来处理 Qt 中尚未定义特定事件处理器的事件。当重新实现 `event()` 函数时, 如果不进行事件处理, 则需要调用基类的 `event()` 函数。

3. 在 `QObject` 中注册事件过滤器

如果对象使用 `installEventFilter()` 函数注册了事件过滤器, 目标对象中的所有事件将首先发给这个监视对象的 `eventFilter()` 函数。

4. 在 `QApplication` 中注册事件过滤器

如果一个事件过滤器被注册到程序中唯一的 `QApplication` 对象, 应用程序中所有对象里的每一个事件都会在它们被送达其他事件过滤器前, 首先抵达这个 `eventFilter()` 函数。它可以用来处理一些通常被 `QApplication` 忽略的事件, 如发送给失效窗口的鼠标事件等。

5. 继承 `QApplication` 并重新实现 `notify()` 函数

Qt 调用 `QApplication` 来发送一个事件。重新实现 `notify()` 函数是在事件过滤器得到所有事件之前获得它们的唯一方法。但事件过滤器使用更为便利, 因为可以同时有多个事件过滤器, 而 `notify()` 函数只有一个。

包括鼠标和键盘事件在内的很多事件都可以被传递。如果事件到达其目标对象之前没有被截获处理, 或者没有被它的目标对象处理, 那么此时, 目标对象的父对象将变成新的目标对象, 整个事件处理过程将重复进行, 直到该事件被处理或到达最顶层对象。

图 11-1 显示了按下鼠标按键事件从对话框的子对象传递到父对象的过程。当用户按下下一个鼠标键时, 这个事件首先被发给当前拥有焦点的窗口部件, 这里假设是右下角的“按钮 4”。如果“按钮 4”

没有处理这个事件, Qt 将该事件发送给父对象“组合框”, 直至最后的“对话框”顶层对象。



图 11-1 对话框中的事件处理过程

11.2 事件处理器

通常事件传递后将调用相应的虚函数, 即事件处理器。例如, 当 `QPaintEvent` 事件送达时会调用 `QWidget::paintEvent()` 函数, 这个虚函数将作为本事件的处理函数做出适当响应, 这里的响应是重绘窗口。 `QWidget` 中的 `event()` 函数将绝大多数常用类型事件传递给特定的事件处理器, 如 `mousePressEvent()`、`keyPressEvent()` 以及 `paintEvent()` 等, 并且忽略了其他类型的事件。

如果用户不想重新实现这些事件处理器, 可以调用 Qt 中其父类已有的实现; 如果用户希望替换原有基类的功能, 则必须重新实现全部相关功能。但是, 前一种方式的适应性不强, 无法满足实际的编程需求。后一种方法往往又比较复杂烦琐, 因此建议采用的方法是对原有基类的事件处理函数进行扩充, 即在已有的事件处理器虚函数中补充相关功能的实现, 不需要改动的地方则可以直接调用基类中的默认实现。例如, 下面的代码在自定义的 `checkbox` 窗口部件中对鼠标左键进行了特殊处理, 而将其他鼠标事件仍然交给其父类 `QCheckBox` 类处理。

```
void MyCheckBox::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        // 处理本类特殊的鼠标左键响应
    } else {
        // 将其他鼠标事件传递给父类
        QCheckBox::mousePressEvent(event);
    }
}
```

偶尔也会遇到这种情况, Qt 没有为某个特定事件提供默认的事件处理器, 或者提供的默认事件处理器完全无法满足用户需求。后一种情况虽然可以通过在子类中重写相关事件处理器来满足需求, 但前一种情况却行不通。此时可以统一采用另一种事件处理方式, 即重新实现 `QObject::event()` 函数。

例如 Tab 键的事件处理。它可以在窗口部件调用 `keyPressEvent()` 之前被 `QWidget::event()` 处理, 用于把焦点传递给焦点链中的下一个窗口部件。这种默认的行为一般情况下是适用的, 但如果希望在文本编辑框中缩进一行, 则需要重新实现 `event()` 函数。此外, 如果事件是用户自定义的, Qt 并未对其定义默认行为, 此时, 也需要重新实现 `event()` 函数。参考实现代码如下:

```
bool MyWidget::event(QEvent *event)
{
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *ke = static_cast<QKeyEvent*>(event);
        if (ke->key() == Qt::Key_Tab) {
```



```

        //处理 Tab 键缩进
        return true;
    }
    else if (event->type() == MyCustomEventType) {
        MyCustomEvent *myEvent = static_cast<MyCustomEvent *>(event);
        // 处理用户自定义事件
        return true;
    }
    return QWidget::event(event);
}

```

如果这个事件是一个键按下，把这个 QEvent 对象转换成一个 QKeyEvent 对象并检查究竟是哪个键被按下。如果是 Tab 键，做一些处理并返回“true”，告知 Qt 已经处理了这个事件。如果这个事件是用户自定义的，就把这个 QEvent 对象转换成用户自定义类型并进行相应处理，然后返回“true”。如果两者都不是，调用其父类的 QWidget::event() 函数进行默认处理。

对于按键的处理 Qt 还提供了一种较高级的处理方式，即使用 QAction。例如，假设在 MainWindow 主窗口中有 newFile() 和 open() 两个槽函数，使用下面的代码来添加按键绑定。

```

MainWindow::MainWindow(QWidget * parent, Qt::WFlags flags)
    : QMainWindow(parent, flags)
{
    QWidget *w = new QWidget;
    setCentralWidget(w);
    newAct = new QAction(tr("&New"), this);
    newAct->setShortcut(tr("Ctrl+N"));
    newAct->setStatusTip(tr("新建文件"));
    connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));
    openAct = new QAction(tr("&Open..."), this);
    openAct->setShortcut(tr("Ctrl+O"));
    openAct->setStatusTip(tr("打开文件"));
    connect(openAct, SIGNAL(triggered()), this, SLOT(open()));
    ....
}

```

正如在本书第一部分（基础部分）看到的，这样处理很容易把某个命令添加到菜单或工具栏中。如果命令没有出现在用户界面中，这些 QAction 对象将被 QShortcut 对象替换，该类在 QAction 内部用来支持键绑定。默认情况下，激活状态下的窗口所包含的窗口部件支持使用 QAction 或 QShortcut 进行键值绑定，可以使用 QAction::setShortcutContext() 或 QShortcut::setContext() 方法修改键值。

需要注意的是选择重新实现 keyPressEvent() 还是使用 QAction，这类似于选择重新实现 resizeEvent() 还是使用 QLayout 子类。如果通过继承 QWidget 来实现一个自定义窗口部件，重新实现一些事件处理器并在其中填写一些功能代码的方式比较直接。但是，如果仅是使用一个窗口部件，直接使用 QAction 和 QLayout 提供的高层接口会更加方便。

下面来看一个具体的鼠标与定时器事件的示例。在这个示例中，用户可以通过鼠标在绘图区任意涂鸦，但每隔 10 秒钟，当前的涂鸦内容将被清空。

```

class DrawArea : public QWidget
{

```

```

    Q_OBJECT
public:
    DrawArea(QWidget *parent = 0);
    bool isModified() const { return modified; }
    void clearImage();
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);
private:
    void drawLineTo(const QPoint &endPoint);
    void resizeImage(QImage *image, const QSize &newSize);
    bool modified;
    bool scribbling;
    int myPenWidth;
    QColor myPenColor;
    QImage image;
    QPoint lastPoint;
};

```

首先从 `QWidget` 继承得到一个绘图类 `DrawArea`，用于根据鼠标轨迹进行直线的绘制。这个类主要重新实现了 5 个事件处理器，它们分别是鼠标键按下事件处理器 `mousePressEvent()`，鼠标移动事件处理器 `mouseMoveEvent()`，鼠标键释放处理器 `mouseReleaseEvent()`，窗口重绘事件处理器 `paintEvent()` 和窗口大小改变事件处理器 `resizeEvent()`。私有成员函数 `drawLineTo()` 完成具体的画线功能，`resizeImage()` 函数则用于调整绘图区大小。私有成员变量 `image`、`myPenWidth`、`myPenColor`、`lastPoint`、`scribbling` 和 `modified` 分别记录绘图板、画笔宽度、画笔颜色、上一点鼠标位置、是否正在绘图以及绘图板是否更改。对外开放的函数接口除构造函数外只有 `isModified()` 和 `clearImage()`，分别用来判断当前绘图板是否已被修改和清空绘图板。

由于这一章的重点是介绍事件处理，而绘图操作在第 6 章已做了介绍，因此这里不再赘述绘图代码。下面详细介绍事件处理器函数。

```

void DrawArea::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        lastPoint = event->pos();
        scribbling = true;
    }
}

```

鼠标键按下事件处理函数 `mousePressEvent()` 首先判断是否为鼠标左键，如果是则在私有 `QPoint` 型成员 `lastPoint` 中记录当前鼠标位置，并置成员变量 `scribbling` 为“true”，表示绘图开始。

```

void DrawArea::mouseMoveEvent(QMouseEvent *event)
{
    if ((event->buttons() & Qt::LeftButton) && scribbling)
        drawLineTo(event->pos());
}

```



鼠标移动事件处理函数 `mouseMoveEvent()` 在判断当前鼠标左键已按下且绘图已开始两个条件都为“真”后, 调用 `drawLineTo()` 函数 (该函数代码省略, 请参考示例程序) 绘制从 `lastPoint` 点到当前鼠标位置的一条直线。

```
void DrawArea::mouseReleaseEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton && scribbling) {
        drawLineTo(event->pos());
        scribbling = false;
    }
}
```

鼠标键释放处理函数 `mouseReleaseEvent()` 的处理流程与 `mouseMoveEvent()` 函数类似, 所不同的仅是在直线绘制完成后置 `scribbling` 为“false”, 表示本次绘图结束。

```
void DrawArea::paintEvent(QPaintEvent * event)
{
    QPainter painter(this);
    painter.drawImage(QPoint(0, 0), image);
}
```

窗口重绘事件处理函数 `paintEvent()` 用当前绘图板 `image` 中的内容重绘全部窗口。

```
void DrawArea::resizeEvent(QResizeEvent *event)
{
    if (width() > image.width() || height() > image.height()) {
        int newWidth = qMax(width() + 128, image.width());
        int newHeight = qMax(height() + 128, image.height());
        resizeImage(&image, QSize(newWidth, newHeight));
        update();
    }
    QWidget::resizeEvent(event);
}
```

窗口大小改变事件处理函数 `resizeEvent()` 的处理过程略显复杂。如果改变后的窗口尺寸大于当前绘图板尺寸, 必须重新计算绘图板尺寸, 然后使用 `resizeImage()` 函数 (该函数代码省略, 请参考示例程序) 根据调整后的尺寸新建一个合适的绘图板以替代原有绘图板, 并显式调用窗口刷新。最后调用父类 `QWidget` 的默认处理函数 `resizeEvent()`。

定时器事件放在父窗口 `MainWindow` 中实现, 定义如下:

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow();
protected:
    void closeEvent(QCloseEvent* event);
    void showEvent(QShowEvent* event);
    void hideEvent(QHideEvent* event);
}
```



```

    void timerEvent(QTimerEvent* event);
private:
    bool maybeSave();
    DrawArea *scribbleArea;
    int timeId;
};

```

在主窗口可见时启动一个 10 秒每次的定时器，当主窗口不可见时则关闭这个定时器，定时器事件处理器定时清空当前绘图板。

```

MainWindow::MainWindow()
{
    scribbleArea = new DrawArea(this);
    setCentralWidget(scribbleArea);
    setWindowTitle(tr("鼠标与定时器事件"));
    resize(480, 320);
    timeId = 0;
}

```

构造函数比较简单，只是新建了一个前面介绍的 DrawArea 类，并将其设置成主窗口的中央窗体，然后设置主窗口标题并调整窗口大小，最后将定时器句柄 timeId 初始化为“0”。

```

void MainWindow::showEvent(QShowEvent* event)
{
    timeId = startTimer(10000);
}

```

当主窗口可见时，showEvent()事件处理函数被调用，这里将启动一个 10 秒每次的定时器，并将该定时器句柄保存在 timeId 私有成员变量中。

```

void MainWindow::hideEvent(QHideEvent* event)
{
    killTimer(timeId);
}

```

当主窗口不可见时，hideEvent()事件处理函数被调用，这里将终止以 timeId 为句柄的定时器。

```

void MainWindow::timerEvent(QTimerEvent* event)
{
    if(event->timerId() == timeId)
        scribbleArea->clearImage();
}

```

当定时器事件到来时，timerEvent()事件处理函数被调用。如果 QTimerEvent 事件的句柄与保存在 timeId 中的句柄相等，就清空当前绘图板。顺便提一下，用户可以根据需要启动多个定时器，并一一保存定时器句柄，最后在 timerEvent()事件处理器中根据定时器句柄值处理不同的定时器的响应。

```

void MainWindow::closeEvent(QCloseEvent *event)
{
    if (maybeSave()) {
        event->accept();
    }
}

```



```

    } else {
        event->ignore();
    }
}

```

最后, 当主窗口关闭时, `closeEvent()` 事件处理函数被调用。根据绘图板内容是否改变选择接受或忽略 `QCloseEvent` 事件。需要说明的是, 关闭事件包含了一个用于指示事件接收窗体是否需要关闭的标识, 如果接收窗体接受关闭事件, 窗口将立刻隐藏或销毁, 如果接收窗体忽略该关闭事件, 窗口将不做任何响应 (虽然某些 X11 的窗口管理系统会强制销毁窗体, 但 Qt 并不是这样设计的)。因此, 这里看到的效果是, 如果实例程序运行后用户不做任何绘图操作, 10 秒钟之内程序是无法关闭的。10 秒钟之后, 由于定时器事件处理器会调用 `clearImage()` 函数清空当前绘图板, 导致绘图板变为修改状态, 关闭事件被接受, 程序窗口关闭。同样, 如果用户进行了绘图操作, 绘图板也将变成修改状态, 导致关闭事件被接受, 从而关闭窗口。

11.3 事件过滤器

Qt 事件模型中一项非常强大的功能是一个 `QObject` 实例可以监视另一个 `QObject` 实例中的事件, 实现方法是在目标对象中安装事件过滤器。这样事件过滤器将在事件到达目标对象之前首先获取该事件, 从而起到监视目标对象事件的效果。

现在再看图 11-1, 假设希望通过使用“空格键”把焦点移到下一个 `QPushButton`。对于这种非标准行为的一种解决方法是继承 `QPushButton`, 然后重新实现 `keyPressEvent()` 来调用 `focusNextChild()`, 代码如下:

```

void MyQPushButton::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Space) {
        focusNextChild();
    } else {
        QPushButton::keyPressEvent(event);
    }
}

```

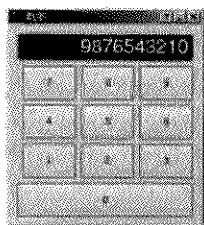


图 11-2 事件过滤器示例

但是这种方法有一个致命缺点, 如果存在各种各样不同的窗口部件, 如 `QComboBox`、`QSpinBox` 等, 就不得不逐个继承它们, 然后分别进行功能扩展。另一个解决方法就是下面要介绍的事件过滤器, 可以在父窗口中监视全部子窗口的事件, 方法是在父对象中注册的过滤器中添加扩展功能的代码。设置事件过滤器需要如下两个步骤:

- 01** 通过对目标对象调用 `installEventFilter()` 来注册监视对象;
- 02** 在监视对象的 `eventFilter()` 函数中处理目标对象的事件。

下面来看一个实例, 如图 11-2 所示, 当用户用鼠标单击显示数字的 `QLineEdit` 时, 已有数字将反色显示 (黑底白字)。

起监视作用的是 `Digital` 类, 它从 `QDialog` 继承而来, 包含 10 个按钮和一个用于显示数字的 `QLineEdit`, 这个 `QLineEdit` 对象即是需要监控的目标对象, 代码如下:

```

class Digital : public QDialog
{
    Q_OBJECT
public:
    Digital(QWidget *parent = 0);
protected:
    bool eventFilter(QObject *target, QEvent *event);
private slots:
    void digitClicked();
private:
    Button *createButton(const QString &text, const QColor &color,
                        const char *member);
    QLineEdit *display;
    enum { NumDigitButtons = 10 };
    Button *digitButtons[NumDigitButtons];
};

```

Digital 类比较容易理解，它包含一个具有 10 个按钮的按钮数组 digitButtons [NumDigitButtons] 和一个用于显示的 QLineEdit 对象。这里的重点是 eventFilter() 事件过滤函数，以及构造函数中事件过滤器的安装。构造函数如下所示：

```

Digital::Digital(QWidget *parent)
    : QDialog(parent)
{
    display = new QLineEdit("");
    display->setReadOnly(true);
    display->setAlignment(Qt::AlignRight);
    display->setMaxLength(15);
    display->installEventFilter(this);
    QFont font = display->font();
    font.setPointSize(font.pointSize() + 8);
    display->setFont(font);
    QColor digitColor(150, 205, 205);
    for (int i = 0; i < NumDigitButtons; ++i) {
        digitButtons[i] = createButton(QString::number(i), digitColor,
                                       SLOT(digitClicked()));
    }
    QGridLayout *mainLayout = new QGridLayout;
    mainLayout->setSizeConstraint(QLayout::SetFixedSize);
    mainLayout->addWidget(display, 0, 0, 1, 4);
    for (int i = 1; i < NumDigitButtons; ++i) {
        int row = ((i - 1) / 3) + 2;
        int column = ((i - 1) % 3) + 1;
        mainLayout->addWidget(digitButtons[i], row, column);
    }
    mainLayout->addWidget(digitButtons[0], 5, 0, 1, 4);
    setLayout(mainLayout);
    setWindowTitle(tr("数字"));
}

```

构造函数完成了界面元素的生成、布局和事件过滤器的注册功能，其中界面元素的生成、布局这里不再赘述。事件过滤器的注册是在 `QLineEdit` 的实例 `display` 中调用 `display->installEventFilter(this)` 方法完成的，该函数将 `Digital` 对象设置成 `display` 对象的事件监视器，而监视代码在 `Digital` 对象的 `eventFilter()` 函数中填写，如下所示：

```
bool Digital::eventFilter(QObject *target, QEvent *event)
{
    if (target == display) {
        if (event->type() == QEvent::MouseButtonPress
            || event->type() == QEvent::MouseButtonDblClick
            || event->type() == QEvent::MouseButtonRelease
            || event->type() == QEvent::ContextMenu) {
            QMouseEvent *mouseEvent = static_cast<QMouseEvent*>(event);
            if (mouseEvent->buttons() & Qt::LeftButton) {
                QPalette newPalette = palette();
                newPalette.setColor(QPalette::Base,
                                   display->palette().color(QPalette::Text));
                newPalette.setColor(QPalette::Text,
                                   display->palette().color(QPalette::Base));
                display->setPalette(newPalette);
            } else {
                display->setPalette(palette());
            }
            return true;
        }
    }
    return QDialog::eventFilter(target, event);
}
```

事件过滤器 `eventFilter` 如果发现当前监控的对象是目标对象 `display`，还需要继续判断监控的事件类型，如果是鼠标的单击、双击、释放或是上下文菜单事件，则进一步判断是否是鼠标左键，如果是，则互换当前 `display` 对象调色板的底色和字体颜色，从而产生反色显示的效果，否则继续使用当前调色板。

最后，如果将 `QApplication` 对象注册成事件过滤器监视对象，那么应用程序中所有对象里的每一个事件都会在他们被送达其他事件过滤器前，首先抵达这个 `eventFilter()` 函数。因此，可以根据需要实现事件的多级过滤。

11.4 加快用户界面响应

细心的读者也许已经发现，所有的实例在 `main()` 最后都调用了 `QApplication::exec()` 函数，它启动了 Qt 的事件循环。开始时 Qt 会发出一些事件来显示和绘制窗口部件。在这之后，事件循环就开始运行，不断地检查是否有事件发生并把这些事件交给 `QObject` 目标对象或其子对象处理。

当一个事件被处理时，其他的事件也可能产生并追加到 Qt 的事件队列中。如果在一个特定事件的处理上消耗过多时间，用户界面就有可能冻结而无法响应。例如，当程序存在大量长时间 I/O 操作时，窗口系统产生的一些事件将无法得到处理直到 I/O 操作完成，从而导致这段时间内用户界面冻结。

在 Qt 中解决上述问题有三种方法：第一种方法是使用线程，一个线程处理应用程序用户界面响应，另一个处理特定事件上的耗时操作，我们在第 10 章已做了介绍；第二种方法是在处理耗时事件时

频繁调用 `QApplication::processEvents()`；第三种方法是推迟耗时事件处理，直到应用程序空闲下来。下面将介绍后两种方法。

11.4.1 使用 `processEvents()` 函数

`QApplication::processEvents()` 函数告知 Qt 处理任何没有被处理的事件，并且将控制权返回给调用者。实际上 `QApplication::exec()` 就是一个不停调用 `processEvents()` 函数的小 `while` 循环。下面我们编写了一个复制本地文件的示例，如图 11-3 所示。

当复制的文件很大时，I/O 操作将持续较长时间，在此期间，程序的其他事件处理将受到影响，如产生界面冻结。为了解决这个问题，在复制函数 `doCopy()` 中调用了 `processEvents()` 来保持用户界面响应，代码如下。

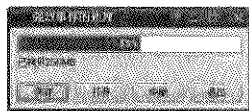


图 11-3 文件复制示例

```
void Dialog::doCopy()
{
    while(bytesToWrite > 0)
    {
        if(isStop){
            rFile->close();
            wFile->close();
            wFile->remove(currentFile);
            progressBar->setMaximum(totalBytes);
            progressBar->setValue(0);
            statusLabel->setText(tr("终止"));
            saveButton->setEnabled(false);
            stopButton->setEnabled(false);
            qApp->processEvents();
            return;
        }else{
            tempBuf = rFile->read(qMin(bytesToWrite, loadSize));
            wFile->write(tempBuf);
            bytesWritten += qMin(bytesToWrite, loadSize);
            bytesToWrite -= qMin(bytesToWrite, loadSize);
            progressBar->setMaximum(totalBytes);
            progressBar->setValue(bytesWritten);
            statusLabel->setText(tr("已拷贝%1MB")
                .arg(bytesWritten / (1024 * 1024)));
            tempBuf.resize(0);
            qApp->processEvents();
        }
    }
    rFile->close();
    wFile->close();
}
```

当还有要复制的数据（`bytesToWrite` 变量值大于 0）且用户没有单击“中断”按钮（`isStop` 变量为 `false`）时，在本次循环中从源文件读取 4K 字节内容并将它写入目标文件，然后更新进度条和状态显



示,最后强制调用 `processEvents()` 函数增加一次处理其他类型事件的机会。如果用户中途中断了本次复制操作,将清理尚未复制完成的文件,更新程序显示状态。这里没有人为地对 `QProgressDialog` 调用 `show()` 函数,因为进度条对话框会自动调用,如果因为某些原因,如文件太小或机器处理速度太快,使得操作很快就完成了, `QProgressDialog` 会检测到这个情况并不再显示。

但是上述方式存在一个潜在的威胁,那就是如果用户在文件复制过程中关闭了主窗口,或者又一次选择了保存按钮,程序的执行结果将是不确定的。对于这个问题,最简单的解决方式是把

```
qApp->processEvents();
```

函数调用替换为

```
qApp->eventLoop()->processEvents(QEventLoop::ExcludeUserInput);
```

函数调用,它告诉 Qt 忽略鼠标和键盘事件。

11.4.2 使用定时器

对于处理长时间操作还有另一种完全不同的方式。这里不是在用户请求中执行处理,而是推迟处理,直到应用程序空闲下来。如果处理可以被安全地打断并且随后可以继续执行,这种方法就能生效,因为无法预先知道多长时间后程序会空闲下来。

在 Qt 中,这种方式可以通过一个特定的“0 毫秒定时器”来实现。只要没有未被处理的事件,这个定时器就会被触发。以下是上述文件复制示例在使用定时器方式下的实现。

```
void Dialog::saveFile()
{
    rFile = new QFile(fileName);
    if (!rFile->open(QFile::ReadOnly) ) {
        QMessageBox::warning(this, tr("程序"),
            tr("无法读取文件 %1:\n%2.",
                .arg(fileName)
                .arg(rFile->errorString())));
        return;
    }

    currentFile = fileName.right(fileName.size() -
        fileName.lastIndexOf('/')-1);
    wFile = new QFile(currentFile);
    if (!wFile->open(QFile::WriteOnly) ) {
        QMessageBox::warning(this, tr("程序"),
            tr("无法写入文件 %1:\n%2.",
                .arg(currentFile)
                .arg(wFile->errorString())));
        return;
    }

    totalBytes = rFile->size();
    bytesToWrite = totalBytes;
    statusLabel->setText(tr("就绪"));
    #ifdef UBET_TIME_EVENT
        timeId = startTimer(0);
```

```

#else
    doCopy();
#endif
}

```

当用户单击“保存”按钮后，saveFile()槽函数开始执行。它首先打开源文件 rFile，然后以同样的文件名在当前目录下建立一个文件 wFile。接下来，以源文件大小初始化总共需复制的字节数变量 totalBytes 和尚未复制的字节数变量 bytesToWrite，并设置当前程序状态为“就绪”。最后是启动一个“0 毫秒的定时器”，这里使用了条件编译方法，如果定义了宏 USET_TIME_EVENT 则开启定时器，否则使用 doCopy() 函数。使用条件编译可以让用户在同一示例程序中选择使用两种加快界面响应的处理方式，这里开启了 USET_TIME_EVENT 宏。

```

void Dialog::timerEvent(QTimerEvent* event)
{
    if(event->timerId() == timeId)
    {
        while((bytesToWrite > 0) && (!qApp->hasPendingEvents()))
        {
            if(isStop){
                rFile->close();
                wFile->close();
                wFile->remove(currentFile);
                progressBar->setMaximum(totalBytes);
                progressBar->setValue(0);
                statusLabel->setText(tr("终止"));
                saveButton->setEnabled(false);
                stopButton->setEnabled(false);
                return;
            }else{
                tempBuf = rFile->read(qMin(bytesToWrite, loadSize));
                wFile->write(tempBuf);
                bytesWritten += qMin(bytesToWrite, loadSize);
                bytesToWrite -= qMin(bytesToWrite, loadSize);
                progressBar->setMaximum(totalBytes);
                progressBar->setValue(bytesWritten);
                statusLabel->setText(tr("已拷贝%1MB")
                    .arg(bytesWritten / (1024 * 1024)));
                tempBuf.resize(0);
            }
        }
        event->accept();
    }
    else{
        Dialog::timerEvent(event);
    }
}

```



函数 `timerEvent()` 用于处理定时器事件，如果 `hasPendingEvents()` 返回 “true”，就停止处理并且把控制权交给 Qt。当 Qt 处理完了它的所有事件，定时器事件的处理将会继续进行。这个函数的其他操作与 `doCopy()` 函数几乎一模一样，所不同的仅是其中没有调用 `processEvents()` 函数。

由于是在空闲的情况下才执行实际操作，定时器处理方式比前面调用 `processEvents()` 函数的方式慢许多，可以很明显的感觉到两者的差异。最后需要说明的是，从 Qt 4.2 开始，事件处理机制采用了 “GTK+ 的底层库 GLib” 为默认的处理方法，而这个处理方法在 Qt 中还有待完善。为了使 `hasPendingEvents()` 函数正常工作，需要设置环境变量 “QT_NO_GLIB=1” 来避免使用 GLib 的事件处理函数。这部分内容有兴趣的读者可以参考附录 D。

11.5 小 结

这一章介绍了 Qt 的事件模型，以及 Qt 对各类事件的处理流程，并详细介绍了在 Qt 程序设计中处理事件的 5 种方法，最后还讨论了如何利用 Qt 事件机制加快用户界面的响应速度。

第12章 数据库

Qt 为数据库访问提供的 QtSql 模块实现了数据库与 Qt 应用程序的无缝集成, 同时为开发人员提供了一套与平台和具体所用数据库均无关的调用接口。这个模块由三部分组成, 如表 12-1 所示, 各部分由不同的 Qt 类支撑。

表 12-1 QtSql 模块层次结构

层 次	描 述
驱动层	驱动层实现了特定数据库与 SQL 接口的底层桥梁, 包括的支持类有: QSqlDriver、QSqlDriverCreator<T>、QSqlDriverCreatorBase、QSqlDriverPlugin 和 QSqlResult
SQL 接口层	SQL 接口层提供了数据库访问类。数据库连接操作由 QSqlDatabase 类提供; QSqlQuery 类提供了与数据库的交互操作; 其他支持类还包括: QSqlError、QSqlField、QSqlTableModel 和 QSqlRecord
用户接口层	用户接口层提供从数据库数据到用于数据表示的窗体的映射。包括的支持类有: QSqlQueryModel、QSqlTableModel 和 QSqlRelationalTableModel。这些类均以 Qt 的模型/视图结构设计

QtSql 模块为不同层次的用户提供了丰富的数据库操作类。对于习惯使用 SQL 语法的用户, QSqlQuery 类提供了直接执行任意 SQL 语句并处理返回结果的方法; 对于那些倾向使用较高层数据库接口而避免使用 SQL 语句的用户, QSqlTableModel 和 QSqlRelationalTableModel 类则提供了合适的抽象。此外, QtSql 模块还支持常用的数据库模式, 如主从视图 (master-detail views) 和向下钻取 (drill-down) 模式。

在阅读这一章前, 读者最好熟悉像 SELECT、INSERT、UPDATE 以及 DELETE 等基本的 SQL 操作, 这将有助于更好地理解本章内容。

12.1 连接数据库

在进行数据库操作前, 必须首先建立与数据库的连接。QtSql 模块使用驱动插件 (driver plugins) 与不同的数据库接口通信。由于 Qt 中 SQL 模块的应用程序接口是与具体数据库无关的, 因此所有与数据库相关的代码均包含在这些驱动插件中。目前, Qt 中支持的驱动插件如表 12-2 所示。

表 12-2 Qt 支持的数据库驱动插件

驱 动	数据库管理系统
QDB2	IBM DB2 及其以上版本
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle Call Interface Driver
QODBC	Open Database Connectivity(ODBC) 包括微软 SQL Server 和其他 ODBC 兼容数据库
QPSQL	PostgreSQL 版本 6.x 和 7.x
QSQLITE	SQLite 版本 3 及其以上版本
QSQLITE2	SQLite 版本 2
QTDS	Sybase Adaptive Server

在使用 QSql 模块之前, 需要对工程进行如下配置:

```
#include <QtSql>
```

Or $\frac{1}{2} = \frac{1}{2}$

```
QSqlDatabase db = QSqlDatabase::addDatabase("OCI", "corina");
db.setHostName("redflag");
db.setDatabaseName("qt430");
db.setUserName("oracle");
db.setPassword("oracle");
db.open();
```

[illegible]

这个例子可以连接当前系统所支持的任意数据库，程序以下拉列表的形式罗列了 Qt 中可用的数据库驱动，具体实现代码如下。

298

```

public:
    ConnDlg(QWidget *parent = 0);
    ~ConnDlg();
    QString driverName() const;
    QString databaseName() const;
    QString userName() const;
    QString password() const;
    QString hostName() const;
    int port() const;
    QSqlError addConnection(const QString &driver, const QString &dbName,
                           const QString &host, const QString &user,
                           const QString &passwd, int port = -1);
    void createSqliteDB();
private slots:
    void on_okButton_clicked();
    void on_cancelButton_clicked() { reject(); }
    void driverChanged(const QString &);
private:
    Ui::QSqlConnectionDialogUi ui;
};

```

类 ConnDlg 中的 addConnection()函数和 createSqliteDB()函数分别完成了对不同数据库的连接。其他函数提供辅助功能，主要负责从用户界面获取用户输入，而用户界面则由 Qt 设计器完成，这里直接引用不再赘述。

```

ConnDlg::ConnDlg(QWidget *parent)
    : QDialog(parent)
{
    ui.setupUi(this);
    QStringList drivers = QSqlDatabase::drivers();
    ui.comboDriver->addItems(drivers);
    connect(ui.comboDriver, SIGNAL(currentIndexChanged( const
        QString & )), this, SLOT(driverChanged(const QString &)));
    ui.status_label->setText(tr("准备连接数据库!"));
}

```

构造函数完成了两项任务：其一是初始化 ui 界面，另一项任务是查找当前所有可用的 Qt 数据库驱动，并将其加入 ui 界面的驱动组合框中。查找数据库驱动的操作由静态函数 QSqlDatabase::drivers() 完成，它以 QStringList 的形式返回所有可用驱动名。将这些驱动名加入 ui 界面的组合框，然后关联这个组合框的 currentIndexChanged(const QString &)信号到槽函数 driverChanged(const QString &)，这样每当用户在这个组合框中选取了不同的驱动时，槽函数就会被调用。最后设置当前程序运行状态。

```

void ConnDlg::driverChanged(const QString & text)
{
    if(text == "SQLITE")
    {
        ui.editDatabase->setEnabled(false);
        ui.editUsername->setEnabled(false);
    }
}

```



```
        ui.editPassword->setEnabled(false);
        ui.editHostname->setEnabled(false);
        ui.portSpinBox->setEnabled(false);
    }
    else{
        ui.editDatabase->setEnabled(true);
        ui.editUsername->setEnabled(true);
        ui.editPassword->setEnabled(true);
        ui.editHostname->setEnabled(true);
        ui.portSpinBox->setEnabled(true);
    }
}
```

槽函数 `driverChanged()` 在发现用户选择的数据库驱动为 `QSQLITE` 时将禁用数据库名、用户名、密码、主机名和端口。这是因为与之对应的 `sqlite` 数据库是一种进程内的本地数据库，无须使用上述特性。后面在 `creatSqliteDB()` 函数中还会继续介绍。

```
void ConnDlg::on_okButton_clicked()
{
    if (ui.comboDriver->currentText().isEmpty()) {
        ui.status_label->setText(tr("请选择一个数据库驱动!"));
        ui.comboDriver->setFocus();
    } else if (ui.comboDriver->currentText() == "QSQLITE"){
        creatSqliteDB();
        accept();
    } else {
        QSqlError err = addConnection(driverName(), databaseName(), hostName(),
                                     userName(), password(), port());
        if (err.type() != QSqlError::NoError)
            ui.status_label->setText(err.text());
        else
            ui.status_label->setText(tr("连接数据库成功!"));
        accept();
    }
}
```

当用户单击了“连接”按钮时，函数 `on_okButton_clicked()` 被调用。这个函数首先检查用户是否选择了一个数据库驱动，然后根据驱动类型进行处理。如果是 `QSQLITE` 则调用 `creatSqliteDB()` 函数创建一个内存数据库；否则，调用 `addConnection()` 函数建立一个其他所选类型数据库的连接，并在出错时显示错误信息，没有错误时在状态栏显示数据库连接成功消息。

```
QSqlError ConnDlg::addConnection(const QString &driver, const QString
                                &dbName, const QString &host, const QString &user,
                                const QString &passwd, int port)
{
    static int cCount = 0;
    QSqlError err;
    QSqlDatabase db = QSqlDatabase::addDatabase(driver,
        QString("conn%1").arg(++cCount));
```

```

db.setDatabaseName(dbName);
db.setHostName(host);
db.setPort(port);
if (!db.open(user, passwd)) {
    err = db.lastError();
    db = QSqlDatabase();
    QSqlDatabase::removeDatabase(QString("conn%1").arg(cCount));
}
return err;
}

```

addConnection()函数用来建立一条数据库连接,连接名使用的是 connX (X=1,2,3...)。操作步骤前面已经进行了解释,这里仅增加了错误处理代码。使用 QSqlError 类处理连接错误,它能提供与具体数据库相关的错误信息。当数据库打开失败时,记录最后的错误,然后引用默认数据库连接,并删除刚才打开失败的连接,最后返回这个错误信息。

```

void ConnDlg::createSqliteDB()
{
    QSqlDatabase::database("in_mem_db", false).close();
    QSqlDatabase::removeDatabase("in_mem_db");
    QSqlDatabase db = QSqlDatabase::addDatabase("SQLITE", "in_mem_db");
    db.setDatabaseName(":memory:");
    if (!db.open())
    {
        ui.status_label->setText(db.lastError().text());
        return;
    }
    QSqlQuery q("", db);
    q.exec("drop table Names");
    q.exec("create table Names (id integer primary key, Firstname varchar,
        Lastname varchar, Work varchar)");
    q.exec(tr("insert into Names values (0, '赵', '匡胤', '皇帝')"));
    q.exec(tr("insert into Names values (1, '钱', '二', '小兵')"));
    q.exec(tr("insert into Names values (2, '孙', '三', '小兵')"));
    q.exec(tr("insert into Names values (3, '李', '四', '小兵')"));
    q.exec(tr("insert into Names values (4, '周', '五', '小兵')"));
    ui.status_label->setText(tr("创建 sqlite 数据库成功!"));
}

```

createSqliteDB()函数实现的功能大体上与 addConnection()函数相同。它首先关闭并删除已有的数据库连接 in_mem_db,然后以同样的名称打开一条新连接。这里唯一需要注意的是数据库名必须是 ":memory:",它是 sqlite 在建立内存数据库时唯一可用的名字。当成功打开连接后,程序使用 SQL 语句创建了一张表,并插入了几条记录,这部分内容后面将详细介绍。

12.2 常用数据库操作

QtSql 模块中的 QSqlQuery 类提供了一个执行 SQL 语句的接口并且可以遍历执行的返回结果集。较高层次访问数据库的类还有 QSqlQueryModel 和 QSqlTableModel 等,它们无须使用 SQL 语句就可以



进行数据库操作，并且可以很容易地将结果在表格中表示出来。下面首先看一下如何在 Qt 中使用 SQL 语句。

12.2.1 使用 SQL 语句

一旦连接建立，就可以使用 QSqlQuery 执行底层数据库支持的 SQL 语句，所要做的仅是创建一个 QSqlQuery 对象，然后再调用 QSqlQuery::exec() 函数。具体过程如下：

```
QSqlQuery query;
query.exec("SELECT kind, elevaltor FROM automobile WHERE elevaltor >= 120");
```

执行 exec() 调用后，就可以操作返回的结果。

```
while (query.next()) {
    QString kind = query.value(0).toString();
    int elevaltor = query.value(1).toInt();
    qDebug() << elevaltor << kind << endl;
}
```

调用一次 next() 将 QSqlQuery 定位到返回的第一条记录，后续的 next() 调用每次将记录指针后移一条记录，直到记录末尾 next() 返回“false”。如果结果集为空（或查询失败），第一次 next() 调用返回“false”。

value() 函数以 QVariant 类型返回一个字段的值。这些字段从 0 开始，按照 SELECT 语句指定的排列顺序进行编号。QVariant 类包含多种 C++ 和 Qt 数据类型，包括 int 和 QString 类型等。数据库中的不同数据类型被映射成相应的 C++ 和 Qt 数据类型存储于 QVariant 变量中。例如，VARCHAR 被映射成 QString 类型，DATETIME 被映射成 QDateTime 类型。

QSqlQuery 还提供了其他用于操作返回结果集的函数：first()、last()、previous() 和 seek()。这些函数使用方便，但对于某些数据库而言，它们比 next() 调用更慢且更耗内存。当操作大数据集时，一项简单的优化手段是在调用 exec() 前调用 QSqlQuery::setForwardOnly(true)，从而只能用 next() 来操作返回结果。

前面将 SQL 语句作为 QSqlQuery::exec() 的参数，但它同样可以直接传给构造函数，从而使得该语句立即被执行，如：

```
QSqlQuery query("SELECT kind, elevaltor FROM automobile WHERE elevaltor >= 120");
```

还可以通过调用 isActive() 检查执行错误。

```
if (!query.isActive())
    QMessageBox::warning(this, tr("数据库错误!"), query.lastError().text());
```

如果无错误则 query 是激活态，可以调用 next() 操作返回结果。

插入操作与查询操作大体相同，代码如下所示：

```
QSqlQuery query("INSERT INTO automobile(id,attribute,type,kind,
nation,carnumber,elevaltor,distance,oil,temperature) "
"VALUES (100,'四轮','轿车','捷达',21,57191,50,20000,30,50)");
```

操作完成之后，如果调用 numRowsAffected() 函数将返回受 SQL 语句影响的行数，或者返回“-1”表示出错。

如果要插入多条记录,或者避免将值转换成字符串(即正确地转义),可以调用 `prepare()` 函数指定一个包含占位符的 `query`,然后绑定要插入的值。通过使用本地支持或在本地不支持的情况进行模拟,Qt 对所有数据库均可以支持 Oracle 类型的占位符和 ODBC 类型的占位符。如下面示例使用了 Oracle 语法的有名占位符。

```
QSqlQuery query;
query.prepare("INSERT INTO automobile(id,attribute,type,kind,
    nation,carnumber,elevaltor,distance,oil,temperature)"
    "VALUES (:id,:attribute,:type,:kind,
    :nation,:carnumber,:elevaltor,
    :distance,:oil,:temperature)");
query.bindValue(:id, 121);
query.bindValue(:attribute, "四轮");
query.bindValue(:type, "轿车");
query.bindValue(:kind, "富康");
query.bindValue(:nation, 21);
query.bindValue(:carnumber, 57191);
query.bindValue(:elevaltor, 50);
query.bindValue(:distance, 20000);
query.bindValue(:oil, 30);
query.bindValue(:temperature, 50);
query.exec();
```

下面的示例则使用了 ODBC 格式的定位占位符。

```
QSqlQuery query;
query.prepare("INSERT INTO automobil "
    "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
query.bindValue(0, 121);
query.bindValue(1, "四轮");
query.bindValue(2, "轿车");
query.bindValue(3, "富康");
query.bindValue(4, 21);
query.bindValue(5, 57191);
query.bindValue(6, 50);
query.bindValue(7, 20000);
query.bindValue(8, 30);
query.bindValue(9, 50);
query.exec();
```

调用 `exec()` 函数后,可以继续调用 `bindValue()` 或 `addBindValue()` 函数绑定新值,然后再次调用 `exec()` 函数在 `query` 中插入新值。

占位符通常使用包含 non-ASCII 字符或非 non-Latin-1 字符的二进制数据和字符串,而 Qt 在后台均使用 Unicode 字符,无论数据库是否支持 Unicode 编码。对于不支持 Unicode 编码的数据库,Qt 将进行隐式的字符串编码转换。

更新操作和插入操作十分类似,如下所示。

```
QSqlQuery query;
```



```
query.exec("UPDATE automobil SET oil = 30 WHERE id = 121");
```

同样，还可以使用有名占位符和定位占位符来绑定实际的值。

最后，删除操作也十分类似，如下所示。

```
QSqlQuery query;
query.exec("DELETE FROM automobil WHERE id = 121");
```

12.2.2 事务操作

在可使用事务的数据库上，Qt 支持 SQL 事务。通过在表示数据库连接的 `QSqlDatabase` 对象上调用 `transaction()` 函数来发起一个事务操作，调用 `commit()` 或 `rollback()` 函数来结束一个事务操作。下面是一个操作示例。

```
QSqlDatabase::database().transaction();
QSqlQuery query;
query.exec("SELECT id FROM factory WHERE manufactory = '一汽'");
if (query.next()) {
    int factoryId = query.value(0).toInt();
    query.exec("INSERT INTO cars (carid, name, factoryid, year) "
              "VALUES (9, '速腾', factoryId, 2005)");
}
QSqlDatabase::database().commit();
```

`QSqlDatabase::database()` 返回前面 `createConnection()` 函数所生成连接的 `QSqlDatabase` 对象。如果一个事务不能启动，`QSqlDatabase::transaction()` 返回 “false”。某些数据库不支持事务，对于这类数据库，`transaction()`、`commit()` 和 `rollback()` 函数不做任何处理。可以通过在与特定数据库相关联的 `QSqlDriver` 上调用 `hasFeature()` 函数测试该数据库是否支持事务，如：

```
QSqlDriver *driver = QSqlDatabase::database().driver();
if (driver->hasFeature(QSqlDriver::Transactions))
    ...
```

数据库的其他特性同样可以进行测试，包括是否支持 BLOBs (Binary Large Objects)、Unicode 和预处理 (prepared queries)。

由于每连接一次只能处理一个活动的事务，对于希望一次执行多个事务的操作，建立多连接将十分有用。使用多连接时仍然可以有一个未命名的连接，在没有指定连接的情况下将使用该无名连接。

12.2.3 使用 SQL 模型类

除了 `QSqlQuery` 类，Qt 还提供了三种用于访问数据库的高层类，它们是 `QSqlQueryModel`、`QSqlTableModel` 和 `QSqlRelationalTableModel` 类，其各自的用途如表 12-3 所示。

表 12-3 高层数据库访问类

类 名	用 途
<code>QSqlQueryModel</code>	基于任意 SQL 语句的只读模型
<code>QSqlTableModel</code>	基于单个表的读写模型
<code>QSqlRelationalTableModel</code>	<code>QSqlTableModel</code> 的子类，增加了外键支持

这三个类均从 `QAbstractTableModel` 类继承,在不涉及数据的图形表示时可以单独使用,进行数据库操作,也可以作为数据源将数据库内的数据在 `QListView` 或 `QTableView` 等基于视图模式的 Qt 类中表示出来。使用它们的另一个好处是,程序员很容易在编程时采用不同的数据源。例如,假设起初打算使用数据库存储数据并使用了 `QSqlTableModel`,后因需求变化决定改用 XML 文件存储数据,程序员此时要做的仅是更换一下数据模型类。下面分别介绍这三个类的使用方法,然后在下一节 Qt 数据库应用中给出一个完整的实例。

`QSqlQueryModel` 类提供了一个只读的数据模型用于表示 SQL 操作的结果。例如:

```
QSqlQueryModel model;
model.setQuery("SELECT * FROM automobile ");
for (int i = 0; i < model.rowCount(); ++i) {
    int carnumber = model.record(i).value("carnumber").toInt();
    QString kind = model.record(i).value("kind").toString();
    qDebug() << kind << carnumber;
}
```

这段代码查询了所有的车辆信息并打印车辆类型和车牌。在使用 `QSqlQueryModel::setQuery()` 进行查询语句的设置后,就可以使用 `record(i)` 函数访问每一条单独的记录了。此外, `setQuery()` 的重载函数还可以接收一个 `QSqlQuery` 对象作为参数,此时可以利用 `QSqlQuery` 类的某些特性,如进行预操作等。

`QSqlTableModel` 提供了一个可读写的数据库模型用于操作单个 SQL 表。下面的例子使用 `QSqlTableModel` 来执行 SELECT 操作。

```
QSqlTableModel model;
model.setTable("automobile");
model.setFilter("elevator > 120");
model.select();
```

它等价于如下查询:

```
SELECT * FROM automobile WHERE elevator > 120
```

通过 `QSqlTableModel::record()` 获取记录,然后用 `value()` 函数获取各字段来操作以下返回结果集。

```
for (int i = 0; i < model.rowCount(); ++i) {
    QString kind = model.record(i).value("kind").toString();
    int carnumber = model.record(i).value("carnumber").toInt();
    qDebug() << kind << carnumber;
}
```

上面两段代码查询了所有超速车辆(时速大于 120 公里/小时),并打印超速车辆的类型和车牌。

`QSqlRecord::value()` 需指定字段名或字段索引。在操作大数据集时,建议通过索引指定字段。例如:

```
int kindIndex = model.record().indexOf("kind");
int carIndex = model.record().indexOf("carnumber ");
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    QString kind = record.value(kindIndex).toString();
    int carnumber = record.value(carIndex).toInt();
    qDebug() << kind << carnumber;
}
```



可以使用向二维模型中插入数据同样的方法向数据库表中插入一条记录。首先调用 `insertRow()` 函数创建一个新行（记录），然后调用 `setData()` 置每一列（字段）的值。例如：

```
QSqlTableModel model;
model.setTable("automobile ");
int row = 0;
model.insertRows(row, 1);
model.setData(model.index(row, 0), 121);
model.setData(model.index(row, 1), "四轮");
model.setData(model.index(row, 2), "轿车");
model.setData(model.index(row, 3), "富康");
model.setData(model.index(row, 4), 21);
model.setData(model.index(row, 5), 57191);
model.setData(model.index(row, 6), 50);
model.setData(model.index(row, 7), 20000);
model.setData(model.index(row, 8), 30);
model.setData(model.index(row, 9), 50);
model.submitAll();
```

调用 `submitAll()` 函数后，该条记录根据表排序规则，可能移到与指定行所不同的行位置。`submitAll()` 调用在失败时返回“false”。标准模型与 SQL 模型间的最大不同是，SQL 模型必须调用 `submitAll()` 将所做的修改写入数据库。

更新记录时，首先必须定位所要修改记录在 `QSqlTableModel` 中的位置（如使用 `select()`），然后依次取出该记录，修改相应字段，最后写回数据库。

```
QSqlTableModel model;
model.setTable("automobile ");
model.setFilter("id = 121");
model.select();
if (model.rowCount() == 1) {
    QSqlRecord record = model.record(0);
    record.setValue("kind ", "桑塔纳");
    record.setValue("carnumber ", 79120);
    model.setRecord(0, record);
    model.submitAll();
}
```

如果有满足条件的记录，通过 `QSqlTableModel::record()` 来获取该记录。然后用修改后的记录覆盖原记录使得更改生效。

更新同样可以向更新非 SQL 模型那样使用 `setData()` 函数。通过指定行、列，可以获得相应的模型索引。例如：

```
model.select();
if (model.rowCount() == 1) {
    model.setData(model.index(0, 3), "桑塔纳");
    model.setData(model.index(0, 5), 79120);
    model.submitAll();
}
```

删除一条记录与更新一条记录类似。

```
model.setTable("automobile ");
model.setFilter("id = 121");
model.select();
if (model.rowCount() == 1) {
    model.removeRows(0, 1);
    model.submitAll();
}
```

`removeRows()`函数需传入的参数分别是待删除部份首记录的行号和需删除的记录数。下面的例子删除满足条件的所有记录。

```
model.setTable("automobile ");
model.setFilter("elevaltor > 120");
model.select();
if (model.rowCount() > 0) {
    model.removeRows(0, model.rowCount());
    model.submitAll();
}
```

`QSqlRelationalTableModel` 类是对 `QSqlTableModel` 类的扩展，它提供了对外键的支持。外键是一张表中的某个字段与另一张表中的主键间的一一映射。例如，建立如下两张表：

```
QSqlQuery query;
query.exec("create table factory (id int primary key, "
    "manufactory varchar(40), "
    "address varchar(40))");
query.exec(QObject::tr("insert into factory values(1, '一汽', '长春')"));
query.exec(QObject::tr("insert into factory values(2, '二汽', '武汉')"));
query.exec(QObject::tr("insert into factory values(3, '上汽', '上海')"));

query.exec("create table cars (carid int primary key, "
    "name varchar(50), "
    "factoryid int, "
    "year int, "
    "foreign key(factoryid) references factory)");
query.exec(QObject::tr("insert into cars values(1, '奥迪', 1, 1990)"));
query.exec(QObject::tr("insert into cars values(2, '红旗', 1, 1950)"));
query.exec(QObject::tr("insert into cars values(3, '捷达', 1, 1990)"));
query.exec(QObject::tr("insert into cars values(4, '东风', 2, 1960)"));
query.exec(QObject::tr("insert into cars values(5, '富康', 2, 2004)"));
query.exec(QObject::tr("insert into cars values(6, '标致', 2, 2001)"));
query.exec(QObject::tr("insert into cars values(7, '桑塔纳', 3, 1995)"));
query.exec(QObject::tr("insert into cars values(8, '帕萨特', 3, 2000)"));
```

表 `cars` 中有一个表示生产厂家的字段 `factoryid` 指向 `factory` 表中的 `id` 字段，于是称 `factoryid` 是一个外键。当使用普通的 `QSqlTableModel` 模型表示数据库表 `cars`，并使用 `QTableView` 进行显示时（数据表示将在 12.2.4 节介绍），效果如图 12-2 所示。



	id	名称	生产厂家	年份
1	1	奥迪	1	1990
2	2	红旗	1	1990
3	3	捷达	1	1990
4	4	东风	2	1990
5	5	富康	2	2004
6	6	新康	2	2001
7	7	奇瑞旗	3	1995
8	8	林肯特	3	2000

图 12-2 使用 QSqlTableModel 表示具有外键的表 cars

如果使用 QSqlRelationalTableModel 模型，并进行如下设置：

```
QSqlRelationalTableModel rModel;
model->setTable("cars");
model->setRelation(2, QSqlRelation("factory", "id", "manufactory"));
```

其中，setRelation()函数告诉 QSqlRelationalTableModel 模型的 factoryid 字段是 factory 表中 id 字段的外键，但其显示为 factory 表的 manufactory 字段，而不是 id 字段。其在 QTableView 下的显示效果如图 12-3 所示。

	id	名称	生产厂家	年份
1	1	奥迪	一汽	1990
2	2	红旗	一汽	1990
3	3	捷达	一汽	1990
4	4	东风	二汽	1990
5	5	富康	二汽	2004
6	6	新康	二汽	2001
7	7	奇瑞旗	上汽	1995
8	8	林肯特	上汽	2000

图 12-3 使用 QSqlRelationalTableModel 表示具有外键的表 cars

通过图形类显示发现，这时生产厂家字段变成了 factory 表中的厂家名称。使用 QSqlRelationalTableModel 模型可以很容易地实现数据库中的主从视图模式。

12.2.4 数据表示

QSqlQueryModel, QSqlTableModel 和 QSqlRelationalTableModel 类均可以作为数据源在 Qt 的视图类中表示，如 QListView, QTableView 和 QTreeView 等视图类。其中 QTableView 类最适合表示二维的 SQL 操作结果。下例创建一个基于 SQL 数据模型的视图。

```
QTableView *view = new QTableView;
view->setModel(model);
view->show();
```

视图类可以显示一个水平表头和一个垂直表头，水平表头在每一列之上显示一个列名，默认情况下列名就是数据库表的字段名，可以通过 setHeaderData()函数修改列名，例如：

```
model->setHeaderData(0, Qt::Horizontal, QObject::tr("ID"));
model->setHeaderData(1, Qt::Horizontal, QObject::tr("品牌"));
model->setHeaderData(2, Qt::Horizontal, QObject::tr("生产厂家"));
model->setHeaderData(3, Qt::Horizontal, QObject::tr("时间"));
```

垂直表头在每一行的最左侧显示本行的行号。如果程序员调用 `QSqlTableModel::insertRows()` 函数插入了一行, 那么新插入行的行号将被标以星号 “*”。直到程序员调用了 `submitAll()` 函数进行提交或系统进行了自动提交; 如果程序员调用了 `QSqlTableModel::removeRows()` 函数删除了一行, 这一行将被标以感叹号 “!”, 直到提交。

对于可读写的模型类 `QSqlTableModel` 和 `QSqlRelationalTableModel`, 视图允许用户编辑其中的字段, 也可以通过如下调用禁止用户编辑。

```
view->setEditTriggers(QAbstractItemView::NoEditTriggers);
```

还可以将同一个数据模型用于多个视图, 一旦用户通过其中某个视图编辑了数据模型, 其他视图也会立即随之得到更新。

视图中的项也可以使用代理。默认的代理是 `QItemDelegate`, 它可以处理大多数常用数据类型如 `int`、`QString`、`QImage` 等。代理同样可以为视图中的项提供诸如组合框之类的编辑窗口部件, 用户可以从 `QAbstractItemDelegate` 或 `QItemDelegate` 类继承来构建自己的代理类。这部分内容请参见第 15 章“模型/视图结构”。

此外, 虽然 `QSqlTableModel` 对单个表的操作进行了优化并提供了读写支持, 但如果用户希望对任意的数据库操作结果集进行更大程度的灵活控制, 则可以直接从 `QSqlTableModel` 的基类 `QSqlQueryModel` 继承, 并通过重写 `flags()` 函数和 `setData()` 函数将其变成可读写的。下面两个功能函数为查询模型的“字段 3”增加了可写属性, 但保持其他字段只读属性不变。

```
Qt::ItemFlags EditableModel::flags(
    const QModelIndex &index) const
{
    Qt::ItemFlags flags = QSqlQueryModel::flags(index);
    if (index.column() == 3)
        flags |= Qt::ItemIsEditable;
    return flags;
}
```

`flags()` 函数的主要功能是通过一个“位或”操作为相应的 `ItemFlags` 增加可写标记, 该标记 `Qt` 定义为 `Qt::ItemIsEditable`, 然后返回修改后的值。

```
bool EditableModel::setData(const QModelIndex &index,
    const QVariant &value, int role)
{
    if (index.column() != 3)
        return false;
    QModelIndex primaryKeyIndex = QSqlQueryModel::index(index.row(), 0);
    int id = data(primaryKeyIndex).toInt();
    clear();
    bool ok;
    ok = setName(id, value.toString());
}
```



```
refresh();  
return ok;  
}
```

setData()修改了表中“字段3”的内容,然后刷新显示。为了完整起见,下面列出了修改函数 setName()的代码,它执行基本的更新操作。

```
bool EditableModel::setName(int personId, const QString &name)  
{  
    QSqlQuery query;  
    query.prepare("update person set name = ? where id = ?");  
    query.addBindValue(name);  
    query.addBindValue(personId);  
    return query.exec();  
}
```

12.3 Qt 数据库应用

现在,读者已经具备了使用 Qt 进行数据库操作的基本知识,接下来将用两个示例综合应用上述知识。

12.3.1 使用嵌入式数据库

通常采用各种数据库来实现对数据的存储、检索等功能,例如,Oracle、SQL Server、MySQL 等。这些产品除提供基本的查询、删除、添加等功能外,还提供很多高级特性,如触发器、存储过程、数据备份恢复、全文检索功能等。但实际上,很多应用仅利用了这些数据库产品的基本特性,而且在某些特殊场合的应用中,这些数据库明显有一些臃肿。Qt 提供了一种进程内数据库——SQLite。它小巧灵活,无须额外安装配置且支持大部分 ANSI SQL92 标准,是一个轻量级的数据库,概括起来具有以下优点:

(1) SQLite 的设计目的是嵌入式 SQL 数据库引擎,它基于纯 C 语言代码,已经应用到了非常广泛的领域。

(2) SQLite 在需要持久存储时可以直接读写硬盘上的数据文件,在无须持久存储时也可以将整个数据库至于内存中,两者均不需要额外的 server 服务端进程,即 SQLite 是无须独立运行的数据库引擎。

(3) 开放源代码,整个代码少于三万行,有良好的注释和 90% 以上的测试覆盖率。

(4) 少于 250KB 的内存占用 (gcc 编译下)。

(5) 支持视图、触发器、事务,支持嵌套 SQL 功能。

(6) 提供虚拟机用于处理 SQL 语句。

(7) 不需要配置,不需要安装,也不需要管理员。

(8) 支持大部分 ANSI SQL92 标准。

(9) 大部分应用的速度比目前常见的客户端/服务器结构的数据库快。

(10) 编程接口简单易用。

在持久存储的情况下,一个完整的数据库就对应磁盘上面的一个文件,它是一种具备了基本数据库特性的数据文件:同一个数据文件可以在不同机器上面使用,可以在不同字节序的机器间自由共享;最大支持 2TB 数据容量,而且性能仅受限于系统的可用内存;没有其他依赖,可以应用于多种操作系

统平台。

下面将以一个控制台程序的形式,使用 SQLite 数据库完成大批量数据的增、删、改、查操作,并打印其操作耗时。

首先创建一个 Qt 应用程序对象 app,并设置中文显示。

```
QApplication app(argc, argv);
QTextCodec::setCodecForTr(QTextCodec::codecForName("gb18030"));
```

接下来以“QSQLITE”为数据库类型,以“:memory:”为数据库名,在本进程地址空间内创建一个 SQLite 数据库。然后使用 QSqlQuery 类创建数据库表“automobil”,该表具有 10 个字段,后面的操作都针对这个表进行。

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName(":memory:");
if (!db.open()) {
    QMessageBox::critical(0, QObject::tr("无法打开数据库,请阅读 Qt SQL 驱动文档!"),
        QObject::tr("退出请按 Cancel 键"), QMessageBox::Cancel,
        QMessageBox::NoButton);
    return false;
}
QSqlQuery query;
bool bSuccess = query.exec("CREATE TABLE automobil (id INT PRIMARY KEY, "
    "attribute VARCHAR, "
    "type VARCHAR, "
    "kind VARCHAR, "
    "nation INT, "
    "carnumber INT, "
    "elevaltor INT, "
    "distance INT, "
    "oil INT, "
    "temperature INT)");
if(bSuccess)
    qDebug() << QObject::tr("数据库表创建成功! \n");
else
    qDebug() << QObject::tr("数据库表创建失败! \n");
```

一旦表建立成功后,就可以进行相应操作了。为了统计操作耗时,程序启动一个计时器 t,然后向表中插入任意的 10000 条记录,操作成功后打印操作消耗的时间。

```
QTime t;
t.start();
query.prepare("INSERT INTO automobil "
    "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
long records = 10000;
for(int i=0;i<records;i++) {
    query.bindValue(0, i);
    query.bindValue(1, "四轮");
    query.bindValue(2, "轿车");
    query.bindValue(3, "富豪");
```



```

query.bindValue(4, rand()%100);
query.bindValue(5, rand()%10000);
query.bindValue(6, rand()%300);
query.bindValue(7, rand()%200000);
query.bindValue(8, rand()%52);
query.bindValue(9, rand()%100);
bSuccess = query.exec();
if(!bSuccess) {
    QSqlError lastError = query.lastError();
    qDebug() << lastError.driverText()
        <<QString(QObject::tr("插入失败"));
}
)
qDebug()<<QObject::tr("插入 %1 条记录, 耗时: %2 ms")
    ,arg(records).arg(t.elapsed());

```

插入操作完成后, 重启计时器 t, 然后查询表中刚刚插入的 10000 条记录, 并按 id 字段的降序排列, 以及打印操作耗时。

```

t.restart();
bSuccess = query.exec("SELECT * FROM automobil ORDER BY id DESC");
if(bSuccess)
    qDebug()<<QObject::tr("排序 %1 条记录, 耗时: %2 ms")
        .arg(records).arg(t.elapsed());
else
    qDebug()<<QObject::tr("排序失败! ");

```

更新操作与插入操作类似, 仅仅是使用的 SQL 语句不同。需要注意操的是, 作开始不要忘记重启计时器。

```

t.restart();
for(int i=0;i<records;i++) {
    query.clear();
    query.prepare(QString("UPDATE automobil SET "
        "attribute=?, type=?, kind=?, "
        "nation=?, carnumber=?, elevaltor=?, "
        "distance=?, oil=?, temperature=? "
        "WHERE id=%1").arg(i));
    query.bindValue(0, "四轮");
    query.bindValue(1, "轿车");
    query.bindValue(2, "富康");
    query.bindValue(3, rand()%100);
    query.bindValue(4, rand()%10000);
    query.bindValue(5, rand()%300);
    query.bindValue(6, rand()%200000);
    query.bindValue(7, rand()%52);
    query.bindValue(8, rand()%100);
    bSuccess = query.exec();
    if(!bSuccess) {

```



```

        QSqlError lastError = query.lastError();
        qDebug() << lastError.driverText()
                <<QString(QObject::tr("更新失败"));
    }
}

qDebug()<<QObject::tr("更新 %1 条记录,耗时: %2 ms")
        .arg(records).arg(t.elapsed());

```

同样,删除操作也是以重启定时器开始,然后执行删除 id 为 1500 记录的操作,并打印操作耗时。

```

t.restart();
query.exec("DELETE FROM automobil WHERE id=1500");
qDebug()<<QObject::tr("删除一条记录,耗时: %1 ms").arg(t.elapsed());

```

最后看一下这个程序的执行效果,如图 12-4 所示。

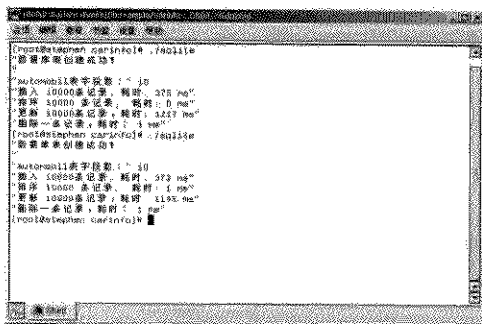


图 12-4 SQLite 实例执行效果

我们会惊奇地发现,SQLite 数据库执行速度非常之快,对 10 000 条记录的各种操作用时几乎均在毫秒一级(除更新操作)。有兴趣的读者可以修改这个实例来测试更大规模的数据库操作,但这将消耗较多的内存空间。

12.3.2 使用 Oracle 数据库

Oracle 数据库是当前使用最广泛的数据库产品之一,Qt 使用 OCI (Oracle Call Interface) 插件提供对 Oracle 数据库的支持。但由于版权的原因,开源版 Qt 并没有提供这个插件。可以使用 Qt 商业版或其评估本来访问 Oracle 数据库,具体步骤如下:

01 首先在 Linux 下安装 Oracle 客户端。可以从 Oracle 网站上下载这个客户端,但它不能在 root 用户下安装,因此可以新建一个用户进行安装。

02 在用户根目录下的 .bashrc 文件中加入 ORACLE_HOME 变量。Oracle 10g 客户端的默认安装路径为 "/opt/oracle/product/10.2.0/client_1",在 .bashrc 中添加如下代码:

```

ORACLE_HOME=/opt/oracle/product/10.2.0/client_1
export ORACLE_HOME

```

03 运行 oemapp console 命令,配置 Oracle 数据库连接。

04 OCI 插件源码由两部分组成，一部分在 `drivers/oci` 目录下，包括 `qsq_oci.cpp` 和 `qsq_oci.h` 两个文件；另一部分在 `sqldrivers/oci` 目录下，包括 `main.cpp` 和 `oci.pro` 两个文件。

05 用 KDevelop 导入工程。在 KDevelop 中选择“工程|导入已有的工程(I)……”，然后选择目录“`QTSRC/src/plugins/sqldrivers/oci`”（QTSRC 为 Qt 源码的路径）

06 配置子工程选项。在工程上单击鼠标右键，选择“子工程选项”，在“Includes”选项卡的“Directories Outside Project”中添加 oracle 客户端目录如“`/opt/oracle/product/10.20/client_1/rdbms/public/`”；在“Libraries”选项卡的“External Libraries:”中添加“`/opt/oracle/oracle/product/10.20/client_1/lib/`”。

07 编译后安装。在 KDevelop 中进行编译，成功后直接在 KDevelop 的 Konsole 中运行 `make install` 进行安装。

完成了 OCI 插件的安装工作之后，接下来看一个较为复杂的示例。它以主从视图的模式展现了汽车制造商与所生产汽车的关系，如图 12-5 所示。

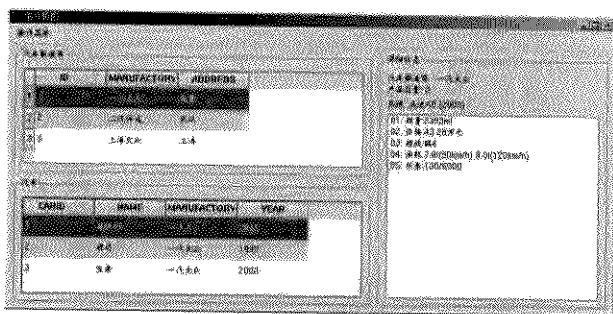


图 12-5 汽车信息示例

在汽车制造商表中选中某一个制造商时，下面的汽车表中将显示该制造商生产的所有产品。在汽车表中选中某个车型时，右边的列表将显示该车型和制造商的详细信息，所不同的是，车型的相关信息存储在 XML 文件中。这个示例综合使用了数据库和 XML 两章的知识。

首先，必须定义相关数据库表，并在其中插入适当信息以供演示使用。所使用的数据库表定义如下：

```
create table factory (id int primary key,
                    manufactory varchar(40),
                    address varchar(40));
insert into factory values(1, '一汽大众', '长春');
insert into factory values(2, '二汽神龙', '武汉');
insert into factory values(3, '上海大众', '上海');
create table cars (carid int primary key,
                  name varchar(50),
                  factoryid int,
                  year int,
                  foreign key(factoryid) references factory);
insert into cars values(1, '奥迪 A6', 1, 2005);
insert into cars values(2, '捷达', 1, 1993);
insert into cars values(3, '宝来', 1, 2000);
```

```

insert into cars values(4, '毕加索', 2, 1999);
insert into cars values(5, '富康', 2, 2004);
insert into cars values(6, '标致 307', 2, 2001);
insert into cars values(7, '桑塔纳', 3, 1995);
insert into cars values(8, '帕萨特', 3, 2000);

```

一些数据库不支持外键, 对于这些数据库, 如果将 FOREIGN KEY 子句剔除, 例子仍然可以工作, 但数据库将不强制执行参照完整性 (enforce referential integrity)。

接下来要做的是建立数据库连接, 这里复用了 12.1 节中的登录对话框, 详细代码请查看示例, 这里不再解释。在示例中提供了 Oracle 连接及备用的 SQLite 连接方式, 以便读者在缺乏 Oracle 数据库支持的时候使用。登录对话框如图 12-6 所示。



图 12-6 “登录”对话框

程序的主窗口定义如下:

```

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(const QString &factoryTable, const QString &carTable,
               QFile *carDetails, QWidget *parent = 0);
private slots:
    void addCar();
    void changeFactory(QModelIndex index);
    void delCar();
    void showCarDetails(QModelIndex index);
    void showFactoryProfile(QModelIndex index);
private:
    QGroupBox *createCarGroupBox();
    QGroupBox *createFactoryGroupBox();
    QGroupBox *createDetailsGroupBox();
    void createMenuBar();
    void decreaseCarCount(QModelIndex index);
    void getAttribList(QDomNode car);
    QModelIndex indexOfFactory(const QString &factory);
    void readCarData();
    void removeCarFromDatabase(QModelIndex index);

```



```
void removeCarFromFile(int id);
QTableView *carView;
QTableView *factoryView;
QListWidget *attribList;
QLabel *profileLabel;
QLabel *titleLabel;
QDomDocument carData;
QFile *file;
QSqlRelationalTableModel *carModel;
QSqlTableModel *factoryModel;
};
```

MainWindow 类定义了程序的主显示界面,包括显示制造商和工厂的两个 QTableView 视图,一个用于显示车型详细信息的 QListWidget 列表,以及相关的信息标签。这个类还实现了对数据库和 XML 文件的操作,接下来将逐一介绍。

MainWindow 类的构造函数实现如下。

```
extern int uniqueCarId;
extern int uniqueFactoryId;
MainWindow::MainWindow(const QString &factoryTable, const QString &carTable,
                      QFile *carDetails, QWidget *parent)
    : QMainWindow(parent)
{
    file = carDetails;
    readCarData();
    carModel = new QSqlRelationalTableModel(this);
    carModel->setTable(carTable);
    carModel->setRelation(2, QSqlRelation(factoryTable, "id", "manufacture"));
    carModel->select();
    factoryModel = new QSqlTableModel(this);
    factoryModel->setTable(factoryTable);
    factoryModel->select();
    QGroupBox *factory = createFactoryGroupBox();
    QGroupBox *cars = createCarGroupBox();
    QGroupBox *details = createDetailsGroupBox();
    uniqueCarId = carModel->rowCount();
    uniqueFactoryId = factoryModel->rowCount();
    QGridLayout *layout = new QGridLayout;
    layout->addWidget(factory, 0, 0);
    layout->addWidget(cars, 1, 0);
    layout->addWidget(details, 0, 1, 2, 1);
    layout->setColumnStretch(1, 1);
    layout->setColumnMinimumWidth(0, 500);
    QWidget *widget = new QWidget;
    widget->setLayout(layout);
    setCentralWidget(widget);
    createMenuBar();
    resize(850, 400);
}
```

```

        setWindowTitle(tr("主从视图"));
    }

```

在构造函数之前声明了两个全局变量 `uniqueCarId` 和 `uniqueFactoryId`，分别用于记录汽车制造商表“factory”和汽车表“cars”的主键。构造函数 `MainWindow()` 需要传入三个参数，分别是汽车制造商表名、汽车表名和读取 XML 文件的 `QFile` 指针。紧接着调用 `readCarData()` 函数将 XML 文件里的车型信息读入 `QDomDocument` 类实例 `carData` 中，以便后面操作。接下来为汽车表“cars”创建一个 `QSqlRelationalTableModel` 模型。`setRelation()` 函数调用告诉该模型的第二个字段（即 cars 表中的 `factoryid` 字段）是 factory 表中 `id` 字段的外键，但其显示为 factory 表的 `manufactory` 字段，而不是 `id` 字段。接着为汽车制造商表“factory”创建一个 `QSqlTableModel` 模型。然后置 `uniqueCarId` 和 `uniqueFactoryId` 全局变量为汽车模型和汽车制造商模型的记录行数，以便后面生成这两个模型的主键。后面进行的是生成界面、菜单，以及界面布置等 ui 操作，这里不再赘述。

```

void MainWindow::changeFactory(QModelIndex index)
{
    QSqlRecord record = factoryModel->record(index.row());
    QString factoryId = record.value("id").toString();
    carModel->setFilter("id = '"+ factoryId +"'");
    showFactorytProfile(index);
}

```

当用户选择了汽车制造商表中的某一行时，`changeFactory()` 槽函数被调用。该函数首先取出用户选择的这条汽车制造商记录，并获取该制造商的主键，然后在汽车表模型 `carModel` 中设置过滤器，使得只显示所选汽车制造商的车型。最后调用 `showFactorytProfile()` 在“详细信息”中显示所选汽车制造商的信息。

```

void MainWindow::showFactorytProfile(QModelIndex index)
{
    QSqlRecord record = factoryModel->record(index.row());
    QString name = record.value("manufactory").toString();
    int count = carModel->rowCount();
    profileLabel->setText(tr("汽车制造商 : %1 \n" \
        "产品数量: %2").arg(name).arg(count));
    profileLabel->show();
    titleLabel->hide();
    attribList->hide();
}

```

函数 `showFactorytProfile()` 分别从汽车制造商模型 `factoryModel` 中获得制造商的名称，从汽车模型 `carModel` 中获得车型数量，然后在“详细信息”的 `profileLabel` 标签中显示这两部分信息。

```

void MainWindow::showCarDetails(QModelIndex index)
{
    QSqlRecord record = carModel->record(index.row());
    QString factory = record.value("manufactory").toString();
    QString name = record.value("name").toString();
    QString year = record.value("year").toString();
    QString carId = record.value("carid").toString();
}

```



```

showFactoryProfile(indexOfFactory(factory));
titleLabel->setText(tr("品牌: %1 (%2)").arg(name).arg(year));
titleLabel->show();
QDomNodeList cars = carData.elementsByTagName("car");
for (int i = 0; i < cars.count(); i++) {
    QDomNode car = cars.item(i);
    if (car.toElement().attribute("id") == carId) {
        getAttribList(car.toElement());
        break;
    }
}
if (!attribList->count() == 0)
    attribList->show();
}

```

当用户继续上一步操作选择了汽车表中的某一行时，showCarDetails()槽函数被调用。该函数首先从汽车模型 carModel 中获取所选记录的所有字段，包括制造商名 factory，车型 name，生产时间 year 和车型主键 carId。然后在详细信息中的 titleLabel 标签中显示该车型的品牌名和生产时间，并重复显示制造商信息。接下来在记录了车型详细信息的 XML 文件中搜索匹配的车辆，这个 XML 文件的格式如下所示：

```

<?xml version="1.0" encoding="gb2312"?>
<archive>
.....
  <car id="5" >
    <attrib number="01" >排量:1600ml</attrib>
    <attrib number="02" >价格:6.58 万元</attrib>
    <attrib number="03" >排放:欧 3</attrib>
    <attrib number="04" >油耗:6.5l(90km/h)</attrib>
    <attrib number="05" >功率:65/5600</attrib>
  </car>
.....
</archive>

```

在 showCarDetails()函数中使用了 QtXml 模块中的 QDomDocument 类完成相关 XML 检索操作。首先找出所有 car 标签，然后在这些标签中找出 id 属性与所选车型主键 carId 相同的属性 id，最后调用 getAttribList()函数显示这个匹配的 car 标签中的相关信息。

```

void MainWindow::getAttribList(QDomNode car)
{
    attribList->clear();
    QDomNodeList attribs = car.childNodes();
    QDomNode node;
    QString attribNumber;
    for (int j = 0; j < attribs.count(); j++) {
        node = attribs.item(j);
        attribNumber = node.toElement().attribute("number");
        QListWidgetItem *item = new QListWidgetItem(attribList);
    }
}

```

```

        QString showText(attribNumber + " " + node.toElement().text());
        item->setText(tr("%1").arg(showText));
    }
}

```

函数 `getAttribList()` 检索上一步中获得的 `car` 标签下的所有子节点,然后将这些子节点的信息在“详细信息”的 `QListWidget` 窗体中显示。这些信息包括信息编号 `number` 和该编号下的信息内容。

```

void MainWindow::delCar()
{
    QModelIndexList selection = carView->selectionModel()->selectedRows(0);
    if (!selection.empty()) {
        QModelIndex idIndex = selection.at(0);
        int id = idIndex.data().toInt();
        QString name = idIndex.sibling(idIndex.row(), 1).data().toString();
        QString factory = idIndex.sibling(idIndex.row(), 2).data().toString();
        QMessageBox::StandardButton button;
        button = QMessageBox::question(this, tr("删除汽车记录"),
                                       QString(tr("确认删除由'%1'生产的'%2'吗? ")
                                              .arg(factory).arg(name)),
                                       QMessageBox::Yes | QMessageBox::No);
        if (button == QMessageBox::Yes) {
            removeCarFromFile(id);
            removeCarFromDatabase(idIndex);
            decreaseCarCount(indexOfFactory(factory));
        }
    } else {
        QMessageBox::information(this, tr("删除汽车记录"),
                                tr("请选择要删除的记录"));
    }
}

```

当用户在菜单中选择了删除操作时, `delCar()` 槽函数被调用。该函数首先判断用户是否在汽车表中选中了一条记录,如果是,将弹出一个确认对话框,提示用户是否删除该记录。得到用户确认后,将依次调用 `removeCarFromFile()` 和 `removeCarFromDatabase()` 函数分别从 XML 文件和数据库表中删除相关内容,最后调用 `decreaseCarCount()` 函数调整汽车制造商表中的内容。如果用户没有在汽车表中选中记录,则提示用户进行选择。

```

void MainWindow::removeCarFromFile(int id)
{
    QDomNodeList cars = carData.elementsByTagName("car");
    for (int i = 0; i < cars.count(); i++) {
        QDomNode node = cars.item(i);
        if (node.toElement().attribute("id").toInt() == id) {
            carData.elementsByTagName("archive").item(0).removeChild(node);
            break;
        }
    }
}

```

函数 `removeCarFromFile()` 遍历 XML 文件中的所有 `car` 标签, 找出 `id` 属性与汽车表中所选记录主键相同的节点, 然后将其删除。

```
void MainWindow::removeCarFromDatabase(QModelIndex index)
{
    carModel->removeRow(index.row());
}
```

函数 `removeCarFromDatabase()` 比较简单, 只需将汽车表中所选中的行从汽车模型 `carModel` 中移除即可, 这个模型将自动删除数据库表中的对应记录。

```
void MainWindow::decreaseCarCount(QModelIndex index)
{
    int row = index.row();
    int count = carModel->rowCount();
    if (count == 0)
        factoryModel->removeRow(row);
}
```

删除了某个汽车制造商的全部产品后, 需要删除这个汽车制造商。函数 `decreaseCarCount()` 判断汽车表中的当前记录数, 如果为零则从汽车制造商表中删除对应的制造商。

```
QModelIndex MainWindow::indexOfFactory(const QString &factory)
{
    for (int i = 0; i < factoryModel->rowCount(); i++) {
        QSqlRecord record = factoryModel->record(i);
        if (record.value("manufactory") == factory)
            return factoryModel->index(i, 1);
    }
    return QModelIndex();
}
```

函数 `indexOfFactory()` 通过制造商的名称进行检索, 并返回一个匹配的模型索引 `QModelIndex`, 以供汽车制造商表模型的其他操作使用。

```
void MainWindow::addCar()
{
    Dialog *dialog = new Dialog(carModel, factoryModel, carData, file, this);
    int accepted = dialog->exec();
    if (accepted == 1) {
        int lastRow = carModel->rowCount() - 1;
        carView->selectRow(lastRow);
        carView->scrollToBottom();
        showCarDetails(carModel->index(lastRow, 0));
    }
}
```

当用户在菜单中选择了添加操作时, `addCar()` 槽函数被调用。它将启动一个添加记录的对话框, 具体添加操作由该对话框完成, 添加完成后进行显示。添加对话框如图 12-7 所示。

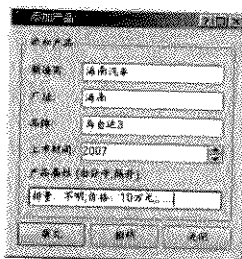


图 12-7 “添加产品”对话框

“添加对话框”定义如下:

```
class Dialog : public QDialog
{
    Q_OBJECT
public:
    Dialog(QSqlRelationalTableModel *cars, QSqlTableModel *factory,
          QDomDocument details,
          QFile *output, QWidget *parent = 0);
private slots:
    void revert();
    void submit();
private:
    int addNewCar(const QString &name, int factoryId);
    int addNewFactory(const QString &factory, const QString &address);
    void addAttribs(int carId, QStringList attribs);
    QDialogButtonBox *createButtons();
    QGroupBox *createInputWidgets();
    int findFactoryId(const QString &factory);
    int generateCarId();
    int generateFactoryId();
    QSqlRelationalTableModel *carModel;
    QSqlTableModel *factoryModel;
    QDomDocument carDetails;
    QFile *outputFile;
    QLineEdit *factoryEditor;
    QLineEdit *addressEditor;
    QLineEdit *carEditor;
    QSpinBox *yearEditor;
    QLineEdit *attribEditor;
};
```

Dialog 类定义了一个“添加对话框”，除界面功能外，它还负责将新加入的记录分别插入汽车制造商表和汽车表，并且将详细的车型信息写入 XML 文件中。具体操作接下来将逐一介绍。

构造函数实现如下：



```
Dialog::Dialog(QSqlRelationalTableModel *cars, QSqlTableModel *factory,
               QDomDocument details, QFile *output, QWidget *parent)
    : QDialog(parent)
{
    carModel = cars;
    factoryModel = factory;
    carDetails = details;
    outputFile = output;
    QGroupBox *inputWidgetBox = createInputWidgets();
    QDialogButtonBox *buttonBox = createButtons();
    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(inputWidgetBox);
    layout->addWidget(buttonBox);
    setLayout(layout);
    setWindowTitle(tr("添加产品"));
}
```

构造函数需要传入汽车表模型、汽车制造商表模型、解析 XML 文件的 QDomDocument 类和读写 XML 文件的 QFile 指针做参数，并将这些参数保存在 Dialog 类的私有变量中。接下来是生成输入界面，以及进行界面布局。

```
void Dialog::submit()
{
    QString factory = factoryEditor->text();
    QString address = addressEditor->text();
    QString name = carEditor->text();
    if (factory.isEmpty() || address.isEmpty() || name.isEmpty()) {
        QString message(tr("请输入厂名、厂址和商品名称！"));
        QMessageBox::information(this, tr("添加产品"), message);
    } else {
        int factoryId = findFactoryId(factory);
        if (factoryId == -1) {
            factoryId = addNewFactory(factory, address);
        }
        int carId = addNewCar(name, factoryId);
        QStringList attribs;
        attribs = attribEditor->text().split(";", QString::SkipEmptyParts);
        addAttribs(carId, attribs);
        accept();
    }
}
```

当用户按下“提交”按钮时，submit()槽函数被调用。这个函数从界面获取用户输入的制造商名 factory、厂址 address 和车型名称 name。如果这三个值中任意一个为空，则以提示框的形式要求用户重新输入。否则，首先调用 findFactoryId()函数在汽车制造商表中查找录入的制造商 factory 的主键 factoryId，如果该主键为“-1”表明录入的制造商不存在，需要调用 addNewFactory()函数插入一条新纪录。接下来调用 addNewCar()函数在汽车表中插入一条新记录。最后从 attribEditor 编辑框中分离出以“分号”间隔的各个属性，将它们保存在 QStringList 列表的 attribs 中，然后调用 addAttribs()函数将

其写入 XML 文件。

```
int Dialog::findFactoryId(const QString &factory)
{
    int row = 0;
    while (row < factoryModel->rowCount()) {
        QSqlRecord record = factoryModel->record(row);
        if (record.value("manufactory") == factory)
            return record.value("id").toInt();
        else
            row++;
    }
    return -1;
}
```

函数 findFactoryId()检索制造商模型 factoryModel 中的全部记录，找出与制造商参数匹配的记录，并将该记录的主键返回。如果未查询到返回“-1”。

```
int Dialog::addNewFactory(const QString &factory, const QString &address)
{
    QSqlRecord record;
    int id = generateFactoryId();
    QSqlField f1("id", QVariant::Int);
    QSqlField f2("manufactory", QVariant::String);
    QSqlField f3("address", QVariant::String);
    f1.setValue(QVariant(id));
    f2.setValue(QVariant(factory));
    f3.setValue(QVariant(address));
    record.append(f1);
    record.append(f2);
    record.append(f3);
    factoryModel->insertRecord(-1, record);
    return id;
}
```

函数 addNewFactory()首先调用 generateFactoryId()函数生成一个汽车制造商表的主键值，然后在汽车制造商表中插入一条新记录，厂名和地址由参数传入。最后返回新记录的主键值。

```
int Dialog::addNewCar(const QString &name, int factoryId)
{
    int id = generateCarId();
    QSqlRecord record;
    QSqlField f1("carid", QVariant::Int);
    QSqlField f2("name", QVariant::String);
    QSqlField f3("factoryid", QVariant::Int);
    QSqlField f4("year", QVariant::Int);
    f1.setValue(QVariant(id));
    f2.setValue(QVariant(name));
    f3.setValue(QVariant(factoryId));
    f4.setValue(QVariant(yearEditor->value()));
}
```



```
record.append(f1);
record.append(f2);
record.append(f3);
record.append(f4);
carModel->insertRecord(-1, record);
return id;
}
```

函数 `addNewCar()` 与函数 `addNewFactory()` 的操作类似。它首先调用 `generateCarId()` 生成一个汽车表的主键值, 然后在汽车表中插入一条新记录, 最后返回这条新记录的主键值。

```
int Dialog::generateFactoryId()
{
    uniqueFactoryId += 1;
    return uniqueFactoryId;
}
int Dialog::generateCarId()
{
    uniqueCarId += 1;
    return uniqueCarId;
}
```

函数 `generateFactoryId()` 和函数 `generateCarId()` 通过将全局变量顺序加一的方式生成一个不重复的主键值, 并将其返回供添加操作使用。

```
void Dialog::addAttribs(int carId, QStringList attribs)
{
    QDomElement carNode = carDetails.createElement("car");
    carNode.setAttribute("id", carId);
    for (int i = 0; i < attribs.count(); i++) {
        QString attribNumber = QString::number(i+1);
        if (i < 10)
            attribNumber.prepend("0");
        QDomText textNode = carDetails.createTextNode(attribs.at(i));
        QDomElement attribNode = carDetails.createElement("attrib");
        attribNode.setAttribute("number", attribNumber);
        attribNode.appendChild(textNode);
        carNode.appendChild(attribNode);
    }
    QDomNodeList archive = carDetails.elementsByTagName("archive");
    archive.item(0).appendChild(carNode);
    if (!outputFile->open(QIODevice::WriteOnly)) {
        return;
    } else {
        QTextStream stream(outputFile);
        archive.item(0).save(stream, 4);
        outputFile->close();
    }
}
```

函数 `addAttribs()` 负责将录入的车型信息写入 XML 文件。该函数首先创建一个 `car` 标签, 然后将 `id` 属性置为传入的车型主键 `carId`, 接下来将每一条信息作为子节点插入, 最后通过输出文件指针 `outputFile` 将修改后的文件写回磁盘。

```
void Dialog::revert()
{
    factoryEditor->clear();
    addressEditor->clear();
    carEditor->clear();
    yearEditor->setValue(QDate::currentDate().year());
    attribEditor->clear();
}
```

函数 `revert()` 的功能十分简单, 它仅撤销用户在界面中的录入信息。这个程序中剩下的功能函数均与用户界面操作有关, 这里省略, 请读者参考实际代码。

最后, 笔者在 Qt 4.2.2 版本、Oracle10g 数据库下测试了这个示例, 它产生如下错误: “QOCIResult::exec: unable to execute select statement: ORA-00933: SQL 命令未正确结束”。原因是在 `MainWindow` 的构造函数中因调用 `setRelation()` 函数而导致下一条 `select()` 函数无法执行。但 Qt 4.3.0 版本已经修复了这个错误, 示例在该版本下运行一切正常。

12.4 小 结

在这一章中, 介绍了 `QtSql` 模块中 SQL 接口层和用户接口层的使用方法, 重点讲解了如何在 Qt 中使用 SQL 语句进行数据库操作和如何利用 `QSqlTableModel` 这类高层次类进行常见的数据库操作, 最后结合两个示例综合演示了 Qt 下的数据库编程方法。`QtSql` 模块虽然为开发者屏蔽了许多数据库信息, 用户在使用高层次进行数据库操作时完全不需要用到 SQL 语句, 但具备基本的数据库知识仍然是必要的。

第 13 章 Qt 的模板库和工具类

Qt 提供了丰富的模板库，包括一些常用的容器（Container）以及基于模板的算法（像 `qFind()`、`qSort()`、`qSwap()` 等）。如果对 C++ 的标准模板库（Standard Template Library, STL）不是很熟悉的话，Qt 的模板库将是一个很好的选择，而且它提供了较详细的说明文档。

Qt 提供了丰富的模板库和工具类。这一章将简要的介绍一些常用的模板库和工具类，希望能够起到抛砖引玉的作用。本章的内容包括 `QString`、`QVariant`、Qt 容器类（Container Classes）、基于模板的算法以及 Qt 正则表达式等。

13.1 Qt 容器类

Qt 提供了一组通用的基于模板的容器类。相比较于 C++ 的标准模板库中的容器类，Qt 的这些容器类更轻量、更安全并且更容易使用。此外，Qt 的容器类在速度、内存消耗和内联（inline）代码（较少的内联代码将会减少可执行程序的大小）等方面进行了优化。

Qt 的容器类为遍历其中的内容提供了两种方法：

（1）Java 风格的迭代器（Java-style iterators）；

（2）STL 风格的迭代器（STL-style iterators），能够同 Qt 和 STL 的通用算法一起使用，并且在效率上也略胜一筹。

Qt 的容器类是可以嵌套的，例如：

```
QHash<QString, QList<double> >
```

其中，`QHash` 的键类型是 `QString`，它的值类型是 `QList<double>`。注意，在最后两个符号“>”之间要保留一个空格，否则的话，C++ 编译器将会把两个“>”符号解释为一个“>>”符号，导致无法通过编译器编译。

存储在 Qt 容器中的数据必须是可赋值的数据类型，也就是说，这种数据类型必须提供一个默认的构造函数（不需要参数的构造函数）、一个复制构造函数和一个赋值操作运算符。这样的数据类型包含了通常使用的大多数数据类型，包括基本数值类型（比如 `int` 和 `double` 等）和 Qt 的一些数据类型（比如 `QString`、`QDate` 和 `QTime` 等）。不过，Qt 的 `QObject` 及其他的子类（比如 `QWidget`、`QDialog` 等）是不能够存储在容器中的，例如：

```
QList<QToolBar> list;
```

上述代码是无法通过编译的，因为这些类（`QObject` 及其他的子类）是没有复制构造函数和赋值操作运算符的。一个可代替的方案是存储 `QObject` 及其子类的指针，例如

```
QList<QToolBar*> list;
```

下面，重点看一下经常使用到的 Qt 容器类。

13.1.1 QList、QLinkedList 和 QVector

`QList<T>`是目前为止最常用的容器类，它存储给定数据类型 `T` 的一系列数值。`QList` 提供了可以在列表进行追加的 `QList::append()`和 `QList::prepend()`函数，也提供了在列表中间完成插入操作的函数 `QList::insert()`。相对于任何其他 Qt 容器类，为了使可执行代码尽可能的少，`QList` 被高度优化。

`QList<T>`维护了一个指针数组，该数组存储的指针指向 `QList<T>`存储的列表项的内容。因此，`QList<T>`提供了基于下标的快速访问。不过，对于不同的数据类型 `QList<T>`采取不同的存储策略：

- 如果 `T` 是一个指针类型或是指针大小的基本类型（即该基本类型占有的字节数和指针类型占有的字节数相同），`QList<T>`会把数值直接存储在它的数组中；
- 其他的情况下，`QList<T>`存储对象的指针，该指针指向实际存储的对象。

下面，看一个例子。

```
// chapter13/container/list/examl.
#include <QDebug>

int main(int argc, char *argv[])
{
    QList<QString> list;
    {
        QString str("This is a test string");
        list << str ;
    }
    qDebug() << list[0] << "How are you!";

    return 0;
}
```

在这个例子中，声明了一个 `QList<QString>` 对象，并通过操作运算符“<<”将一个 `QString` 字符串存储在该列表中。程序中大括弧“{”和“}”括起来的作用域表明，`QList<T>`保存了对象的一个复制。

继承自 `QList` 的类有 `QItemSelection`、`QQueue`、`QSignalSpy` 以及 `QStringList` 和 `QTestEventList`。

`QLinkedList<T>`是一个链式列表，它以非连续的内存块保存数据。`QLinkedList<T>`不能够使用下标而只能使用迭代器访问它的数据项。同 `QList` 比较，当对一个很大的列表进行插入操作时，`QLinkedList` 具有更高的效率。

`QVector<T>`在相邻的内存中存储给定数据类型 `T` 的一组数值。在一个 `QVector` 的前部或者中间位置进行插入操作的速度是很慢的，这是因为这样的操作将会导致内存中的大量数据被移动，这是由 `QVector` 存储数据的方式决定的。`QVector<T>`既可以使用下标访问数据项，也可以使用迭代器访问数据项。继承自 `QVector` 的类有 `QPolygon`、`QPolygonF` 和 `QStack`。

在开发一个较高性能需求的应用程序时，程序员会比较关注这些容器类的运行效率，表 13-1 列出了 `QList`、`QLinkedList` 和 `QVector` 容器的时间复杂度。

表 13-1 时间复杂度的比较

容器类	查找	插入	头部添加	尾部添加
<code>QList</code>	$O(1)$	$O(n)$	Amort. $O(1)$	Amort. $O(1)$
<code>QLinkedList</code>	$O(n)$	$O(1)$	$O(1)$	$O(1)$
<code>QVector</code>	$O(1)$	$O(n)$	$O(n)$	Amort. $O(1)$



注意, “Amort. $O(1)$ ”表示, 如果仅仅完成一次操作, 可能会有 $O(n)$ 行为; 但是如果完成多次操作 (例如 n 次), 平均结果将会是 $O(1)$ 。

1. Java 风格迭代器遍历容器

Java 风格的迭代器是 Qt 4 新加入的一个功能。同 STL 风格的迭代器相比, 它使用起来更简单方便, 不过这也是以轻微的性能损耗为代价的。对于每一个容器类, Qt 都提供了两种类型的 Java 风格迭代器数据类型: 一种提供只读访问, 一种提供读写访问, 其分类如表 13-2 所示。

表 13-2 Java 风格迭代器

容器类	只读迭代器类	读写迭代器类
QList<T>, QQueue<T>	QListIterator<T>	QMutableListIterator<T>
QLinkedList<T>	QLinkedListIterator<T>	QMutableLinkedListIterator<T>
QVector<T>, QStack<T>	QVectorIterator<T>	QMutableVectorIterator<T>

下面以 QList 为例, 分别介绍 Java 风格的两种迭代器。QLinkedList 和 QVector 具有和 QList 相同的遍历接口。

不同于 STL 风格的迭代器, Java 风格迭代器的迭代点 (Java-style iterators point) 位于列表项的中间, 而不是直接指向某个列表项。因此, 它的迭代点或者在第一个列表项的前面, 或者在两个列表项之间, 或者在最后一个列表项之后。如图 13-1 所示。

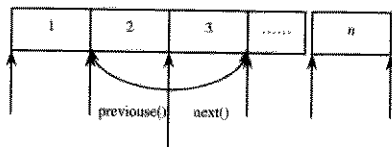


图 13-1 Java 风格的遍历

下面, 看一个实现 QList 只读遍历的例子。

```
// chapter13/container/list/exam2.
#include <QDebug>

int main(int argc, char *argv[])
{
    QList<int> list;
    list << 1 << 2 << 3 << 4 << 5;

    QListIterator<int> i(list);
    for( ; i.hasNext(); )
        qDebug() << i.next();

    return 0;
}
```

这是一个简单的控制台程序。对于 Qt 的一些类, 比如 QString、QList 等, 不需要 QApplication (对于 GUI 用户界面程序则使用 QApplication) 的支持也能够工作, 因此本例没有创建 QApplication

对象。不过，在使用 Qt 编写应用程序的时候，如果是控制台应用程序，笔者建议初始化一个 `QCoreApplication` 对象；而如果是 GUI 图形用户界面程序，则初始化 `QApplication` 对象。头文件 `<QDebug>` 已经包含了 `QList` 的头文件。

下面，看一下这个程序是如何工作的。首先，创建一个 `QList<int>` 栈对象 `list`，并用操作运算符“<<”输入 5 个整数值。接着以该 `list` 为参数初始化一个 `QListIterator` 对象 `i`。此时，迭代点处在第一个列表项“1”的前面（注意，并不是指向该列表项）。然后，调用 `QListIterator<T>::hasNext()` 函数检查当前迭代点之后是否有列表项。如果有，则调用 `QListIterator<T>::next()` 进行遍历。`next()` 函数将会跳过一个列表项（即迭代点将位于第一个列表项和第二个列表项之间），并返回它跳过的列表项的内容。

程序的运行结果为：

```
1 2 3 4 5
```

此外，`QListIterator<T>` 还提供了对列表进行向前遍历的函数。函数 `QListIterator<T>::toBack()` 将迭代点移动到最后一个列表项的后面；函数 `QListIterator<T>::hasPrevious()` 检查当前迭代点之前是否具有列表项；函数 `QListIterator<T>::previous()` 函数返回前一个列表项的内容并将迭代点移动到前一个列表项之前。

`QListIterator<T>` 其他的函数还有：

- `toFront()`，移动迭代点到列表的前端（第一个列表项的前面）。
- `peekNext()`，返回下一个列表项，但不移动迭代点。
- `peekPrevious()`，返回前一个列表项，但不移动迭代点。
- `findNext()`，从当前迭代点开始向后查找指定的列表项，如果找到则返回 `true`，此时迭代点位于匹配列表项的后面；如果没有找到，则返回 `false`，迭代点位于列表的后端（最后一个列表项的后面）。
- `findPrevious()`，与 `findNext()` 类似，不同的是它的方向是先前，查找操作完成后的迭代点在匹配项的前面或整个列表的前端。

`QListIterator<T>` 是只读迭代器，它不能够完成列表项的插入和删除操作。读写迭代器 `QMutableListIterator<T>` 除了提供基本的遍历操作（与 `QListIterator` 的操作相同），还提供了 `insert()` 插入操作函数、`remove()` 删除操作函数和修改数据函数等。

下面看一个例子。

```
// chapter13/container/list/exam3.
#include <QDebug>

int main(int argc, char *argv[])
{
    QList<int> list;
    QMutableListIterator<int> i(list);
    for(int j=0; j<10; ++j)
        i.insert(j);
```

创建一个空的列表 `list` 和该列表的读写迭代器。通过 `QMutableListIterator<T>::insert()` 插入操作为该列表插入 10 个整数值。

```
for(i.toFront(); i.hasNext(); )
    qDebug() << i.next();
```



将迭代器的迭代点移动到列表的前端, 完成对列表的遍历和输出。

```
for(i.toBack(); i.hasPrevious();)
{
    if(i.previous() % 2 == 0)
        i.remove();
    else
    {
        i.setValue(i.peekNext() * 10);
    }
}
```

移动迭代器的迭代点到列表的后端, 对列表进行遍历。如果前一个列表项的值为偶数, 则将该列表项删除; 否则, 将该列表项的值修改为原来的 10 倍。

函数 `QMutableListIterator<T>::setValue()` 修改遍历函数 `next()`、`previous()`、`findNext()` 和 `findPrevious()` 跳过的列表项的值, 但不会移动迭代点的位置。对于 `findNext()` 和 `findPrevious()` 有些特殊: 当 `findNext()` (或 `findPrevious()`) 查找到的时候, `setValue()` 将会修改匹配的列表项; 如果没有找到的话, 对 `setValue()` 的调用将不会做任何修改。

```
for(i.toFront(); i.hasNext();)
    qDebug() << i.next();

return 0;
}
```

最后, 重新遍历并输出列表。程序运行结果如下:

```
0  1  2  3  4  5  6  7  8  9
10 30 50 70 90
```

2. STL 风格迭代器遍历容器

对 STL 风格迭代器的支持始于 Qt 2.0。STL 风格的迭代器兼容 Qt 和 STL 的通用算法, 并在速度上进行了优化。

对于每一个容器类, Qt 都提供了两种类型的 STL 风格的迭代器, 只读访问和读写访问, 分类如表 13-3 所示。应尽可能地使用只读类型的迭代器, 因为它们要比读写迭代器速度更快。

表 13-3 STL 风格迭代器

容器类	只读迭代器类	读写迭代器类
<code>QList<T></code> , <code>QQueue<T></code>	<code>QList<T>::const_iterator</code>	<code>QList<T>::iterator</code>
<code>QLinkedList<T></code>	<code>QLinkedList<T>::const_iterator</code>	<code>QLinkedList<T>::iterator</code>
<code>QVector<T></code> , <code>QStack<T></code>	<code>QVector<T>::const_iterator</code>	<code>QVector<T>::iterator</code>

STL 风格的迭代器的 API 是建立在指针操作的基础上的。例如, `++` 操作运算符移动迭代器到下一项 (item), 而 `*` 操作运算符返回迭代器指向的项。

下面以 `QList` 为例, 来讨论 STL 风格迭代器的应用。`QLinkedList` 和 `QVector` 容器的迭代器接口完全和 `QList` 相同。

不同于 Java 风格的迭代器, STL 风格的迭代器的迭代点直接指向列表项, 如图 13-2 所示。

- `QList<T>::begin()` 函数，返回指向第一个列表项的迭代器；
- `QList<T>::end()` 函数，返回一个容器最后列表项之后的虚拟列表项的、标记无效位置的迭代器，它用于判断是否到达容器的底部。

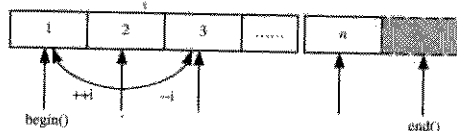


图 13-2 STL 风格的遍历

下面，看一个 STL 风格迭代器应用的例子。

```
// chapter13/container/list/exam4.
#include <QDebug>

int main(int argc, char *argv[])
{
    QList<int> list;
    for(int j=0; j<10; j++)
        list.insert(list.end(), j);
```

初始化一个空的 `QList<int>` 列表，并使用 `QList<T>::insert()` 函数插入 10 个整数值。此处调用的函数 `QList<T>::insert()` 具有两个参数，第 1 个是 `QList<T>::iterator` 类型的参数，表示在该列表项之前插入一个新的列表项（使用 `QList<T>::end()` 函数返回的迭代器，表示在列表的最后插入一个列表项）；第 2 个参数指定了需要插入的值。

```
QList<int>::iterator i;
for(i=list.begin(); i!=list.end(); ++i)
{
    qDebug() << (*i);
    *i = (*i) * 10;
}
```

初始化一个 `QList<int>::iterator` 读写迭代器，在控制台输出列表的同时将列表的所有值增大 10 倍。

```
QList<int>::const_iterator ci;
for(ci=list.constBegin(); ci!=list.constEnd(); ++ci)
    qDebug() << *ci;

return 0;
}
```

初始化一个 `QList<int>::const_iterator` 读写迭代器，在控制台输出列表的所有值。

编译、运行应用程序，输出结果如下：

```
0 1 2 3 4 5 6 7 8 9
0 10 20 30 40 50 60 70 80 90
```



13.1.2 QMap、QHash

`QMap<Key, T>`提供了一个从类型为 `Key` 的键到类型为 `T` 的值的映射。通常, `QMap` 存储的数据形式是一个键对应一个值, 并且按照键 `Key` 的次序存储数据。不过为了能够支持一键多值的情况, `QMap` 提供了 `QMap<Key, T>::insertMulti()` 和 `QMap<Key, T>::values()` 函数。存储一键多值的数据时, 也可以使用 `QMultiMap<Key, T>` 容器, 它继承自 `QMap`。

`QHash<Key, T>` 具有和 `QMap` 几乎完全相同的 API。 `QHash` 维护着一张哈希表 (hash table), 哈希表的大小和 `QHash` 的数据项的数目相适应。 `QHash` 以任意的顺序组织它的数据。当存储数据的顺序无关紧要时, 建议使用 `QHash` 作为存放数据的容器。 `QHash` 也可以存储一键多值形式的数据, 它的子类 `QMultiHash<Key, T>` 实现了一键多值的语义。

`QMap` 和 `QHash` 具有非常类似的功能, 差别是:

- (1) `QHash` 具有比 `QMap` 更快的查找速度;
- (2) `QHash` 以任意的顺序存储数据项, 而 `QMap` 总是按照键 `Key` 顺序存储数据;
- (3) `QHash` 的键类型 `Key` 必须提供 `operator==()` 和一个全局的 `qHash(Key)` 函数, 而 `QMap` 的键类型 `Key` 必须提供 `operator<()` 函数。

二者的时间复杂度如表 13-4 所示。

表 13-4 时间复杂度的比较

容器类	键查找		插入	
	平均	最坏	平均	最坏
<code>QMap</code>	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
<code>QHash</code>	Amort. $O(1)$	$O(n)$	Amort. $O(1)$	$O(n)$

“Amort. $O(1)$ ” 的含义同表 13-1。

下面以 `QMap<T>` 为例, 看一下容器 `QMap<T>` 和 `QHash<T>` 的应用。

1. Java 风格迭代器遍历容器

如表 13-5 所示。

表 13-5 Java 风格迭代器

容器类	只读迭代器类	读写迭代器类
<code>QMap<Key, T>, QMultiMap<key, T></code>	<code>QMapIterator<Key, T></code>	<code>QMutableMapIterator<Key, T></code>
<code>QHash<Key, T>, QMultiHash<Key, T></code>	<code>QHashIterator<Key, T></code>	<code>QMutableHashIterator<key, T></code>

下面是一个在 `QMap` 中插入、遍历和修改的例子。

```
// chapter13/container/map/exam1.
#include <QDebug>

int main(int argc, char *argv[])
{
    QMap<QString, QString> map;
    map.insert("beijing", "111");
```

```
map.insert("shanghai", "021");
map.insert("tianjin", "022");
map.insert("chongqing", "023");
map.insert("jinan", "0531");
map.insert("wuhan", "027");
```

创建一个 QMap 栈对象，并插入<城市，区号>对。

```
QMapIterator<QString, QString> i(map);
for( ; i.hasNext(); )
    qDebug() << " " << i.peekNext().key() << " "
               << i.next().value();
```

通过创建一个只读迭代器，完成对 QMap 的遍历输出。

注意，在输出 QMap 的键和值时，调用的函数是不同的。因为，在输出键的时候，还不需让迭代点移动到下一个位置，因此调用了 QMapIterator<T, T>::peekNext(); 而在输出值的时候调用了 QMapIterator<T, T>::next()。

```
QMutableMapIterator<QString, QString> mi(map);
if(mi.findNext("111"))
    mi.setValue("010");
```

查找某个<键，值>对，然后修改值。Java 风格的迭代器没有提供查找键的函数。因此，在例子中通过查找值的函数 QMutableMapIterator<T, T>::findNext()来实现查找和修改。如果读者有兴趣的话，自己可以实现查找键的功能。

```
QMapIterator<QString, QString> modi(map);
qDebug() << " ";
for( ; modi.hasNext(); )
    qDebug() << modi.peekNext().key() << modi.next().value();

return 0;
}
```

最后，再次遍历并输出修改后的结果。程序运行结果如下：

```
"beijing"          "111"
"chongqing"        "023"
"jinan"            "0531"
"shanghai"         "021"
"tianjin"           "022"
"wuhan"            "027"

"beijing"          "010"
"chongqing"        "023"
"jinan"            "0531"
"shanghai"         "021"
"tianjin"           "022"
"wuhan"            "027"
```

从上面的结果可以看出，QMap<T, T>是按照键的顺序键存储数据的。



2. STL 风格迭代器遍历容器

表 13-6 STL 风格迭代器

容器类	只读迭代器类	读写迭代器类
QMap<Key, T>, QMultiMap<key, T>	QMap<Key, T>::const_iterator	QMap<Key, T>::iterator
QHash<Key, T>, QMultiHash<Key, T>	QHash<Key, T>::const_iterator	QHash<Key, T>::iterator

下面程序的功能和使用 Java 风格迭代器的例子基本相同。不同的是，这里通过查找键来实现值的修改。

```
// chapter13/container/map/exam2.
#include <QDebug>

int main(int argc, char *argv[])
{
    QMap<QString, QString> map;
    map.insert("beijing", "111");
    map.insert("shanghai", "021");
    map.insert("tianjin", "022");
    map.insert("chongqing", "023");
    map.insert("jinan", "0531");
    map.insert("wuhan", "027");

    QMap<QString, QString>::const_iterator i;
    for( i=map.constBegin(); i!=map.constEnd(); ++i)
        qDebug() << i.key() << " " << i.value();

    QMap<QString, QString>::iterator mi;
    mi = map.find("beijing");
    if(mi != map.end())
        mi.value() = "010";
}
```

注意，我们将新的值直接赋给 QMap<QString, QString>::iterator::value()返回的结果，因为该函数返回的是<键, 值>对中的值的引用。

```
QMap<QString, QString>::const_iterator modi;
qDebug() << " ";
for( modi=map.constBegin(); modi!=map.constEnd(); ++modi)
    qDebug() << modi.key() << " " << modi.value();
return 0;
}
```

最后，编译运行程序。程序的输出结果和上面的程序完全相同的。

13.2 QString

字符串是程序员编程时经常使用的数据类型之一，任何一个 Qt GUI 应用程序似乎也无以例外的

都要使用到字符串。标准 C++ 提供了两种字符串，一种是 C 风格的以 “\0” 字符结尾的字符数组，一种是字符串类 `string`。而 Qt 字符串类 `QString` 提供了更强大的功能。

`QString` 类保存 16 位 Unicode 值，提供了丰富的操作、查询、转换等函数。

此外，`QString` 类进行了多方面的优化，包括使用隐式共享（implicit sharing）和高效的内存分配策略。

13.2.1 隐式共享

程序在处理共享对象时，有两种方法复制对象：深拷贝和浅拷贝。所谓深拷贝，就是生成对象的一个完整的复制品；而浅拷贝则是一个引用复制（比如，仅仅复制指向共享数据的指针）。显然，执行一个深拷贝的代价是比较昂贵的，要占有更多的内存和 CPU 资源；而浅拷贝的效率则很好，它仅仅需要设置一个指向共享数据块的指针以及修改引用计数的值。

`QString` 类采用隐式共享技术，将深拷贝和浅拷贝有机地结合起来。

隐式共享又叫回写复制（copy on write）。当两个对象共享同一份数据时（通过浅拷贝实现数据块的共享），如果数据不改变，不进行数据的复制。而当某个对象需要改变数据时，则执行深拷贝。

下面，通过一个例子来说明隐式共享是如何工作的。

```
QString str1 = "data";
QString str2 = str;    // str2: "data"
```

初始化一个内容为 “data” 的字符串，并且将该字符串对象 `str1` 赋值给另一个字符串 `str2`（由 `QString` 的拷贝构造函数完成 `str2` 的初始化）。在对 `str2` 赋值的时候，将发生一次浅拷贝，导致两个 `QString` 对象都指向同一个数据结构。该数据结构除了保存字符串 “data” 以外，还保存了一个引用计数器，以记录字符串数据的引用次数。因为 `str1` 和 `str2` 指向同一个数据结构，因此计数器的值为 2。

```
str2[3] = 'e';    //str2: "date", str1: "data"
```

这个对 `QString` 对象 `str2` 的修改，将会导致一次深拷贝，使得 `str2` 对象指向一个新的、不同于 `str1` 所指的数据结构（该数据结构的引用计数为 1，因为只有 `str2` 指向这个数据结构），同时修改原来的、`str1` 指向的数据结构，设置它的引用计数为 1（此时，只有 `QString` 对象 `str1` 指向该数据结构）。继续在这个 `str2` 指向的、新的数据结构上完成数据的修改。引用计数为 1 意味着这个数据没有被共享。

```
str2[0] = 'f';    // str2: "fate", str1: "data"
```

进一步对 `QString` 对象 `str2` 进行修改，但这个操作不会引起任何形式的拷贝，因为 `str2` 指向的数据结构没有被共享。

```
str1 = str2;
```

将 `str2` 赋值给 `str1`。此时，`str1` 修改它指向的数据结构的引用计数器的值为 0，也就是说没有 `QString` 对象再使用这个数据结构了。因此，`str1` 指向的数据结构将会从内存中释放掉。该操作的结果是，`QString` 对象 `str1` 和 `str2` 都指向字符串为 “fate” 的数据结构，该数据结构的引用计数为 2。

隐式共享的好处是显而易见的，它可以降低对内存和 CPU 资源的使用，提高程序的运行效率。它使得在函数中（比如参数、返回值）使用值传递更有效率。

Qt 支持隐式共享的类，还有：

- 所有的容器类；
- `QByteArray`, `QBitmap`, `QBrush`, `QByteArray`, `QCursor`, `QDir`, `QFont`, `QImage`, `QPen`, `QPalette`,



QPixmap 和 QVariant 等。

13.2.2 内存分配策略

QString 在一个连续的内存块中保存字符串数据。当字符串的大小不断增长的时候, QString 需要重新分配内存空间, 以足以保存增加的字符串。QString 使用的内存分配策略是:

- 每次分配 4 个字符空间, 直到大小为 20。
- 在 20~4084 之间, QString 分配的内存块大小以 2 倍的速度增长。具体地讲, 就是上一次分配的内存块大小的两倍再加 12。
- 从 4084 开始, 每次以 2048 个字符大小 (4096 个字节, 即 4KB) 的步长增长。

下面来看一个例子, 看一下 QString 在后台到底是怎么做的。

```
QString test()
{
    QString str;
    for(int i=0; i<9000; ++i)
        str.append("a");

    return str;
}
```

定义了一个 QString 栈对象 str, 然后为它追加 9000 个字符。根据 QString 的内存分配策略, 这个循环操作将会导致 14 次内存重分配: 4、8、16、20、52、116、244、500、1012、2036、4084、6132、8180、10228。最后的一次内存重分配操作后, QString 对象 str 具有一个 10228 个 Unicode 字符大小的内存块 (20456 字节), 其中有 9000 个字符空间被使用 (18000 字节)。

13.2.3 操作字符串

QString 提供了一个二元的 + 操作符组合两个字符串, 并且提供了一个 += 操作符将一个字符串追加到另一个字符串的末尾。例如,

```
QString str1 = "How " ;
str1 = str1 + "are you !"; //str1: "How are you !";
QString str2 = "Fine, " ;
str2 += "thank you."; // str2: "Fine, thank you";
```

第 1 行代码传递给 QString 一个 const char* 类型的 ASCII 字符串 "How", 它被解释为一个典型的以 "\0" 结尾的 C 类型字符串。这将会引起对 QString 的构造函数

```
QT_ASCII_CAST_WARN_CONSTRUCTOR QString::QString(const char* str)
```

的调用, 来初始化一个 QString 字符串。被传递的 const char* 类型的指针又将被函数 QString::fromAscii() 转化为 Unicode 编码。默认情况下, QString::fromAscii() 函数会把超过 128 的字符作为 Latin-1 进行处理 (不过, 可以通过调用 QTextCodec::setCodecForCStrings() 函数改变 QString::fromAscii() 函数的处理方式。此处不再详细叙述, 有兴趣的读者可以查看 Qt 提供的帮助文档)。此外, 在编译应用程序的时候, 也可以通过定义 QT_CAST_FROM_ASCII 宏变量屏蔽掉该构造函数。如果程序员要求显示给用户的字符串都必须经过 QObject::tr() 函数的处理, 那么屏蔽掉 QString 的这个构造函数是非常有用的。

QString::append() 函数具有和 += 操作符一样的功能, 实现在一个字符串的末尾追加另一个字符串。


```
QString str1 = "How ";
QString str2 = "are ";
str1.append(str2); // str1: "How are ";
str1.append("you !"); // str1: "How are you!";
```

组合字符串的另一个函数是 `QString::sprintf()`,

```
QString str;
str.sprintf("%s", "How "); // str: "How ";
str.sprintf("%s", "are you!"); // str: "are you!";
str.sprintf("%s %s", "How ", "are you!"); // str: "How are you!";
```

`QString::sprintf()` 函数支持的格式定义符和 C++ 库中的函数 `sprintf()` 定义的一样。

此外, Qt 还提供了另一种方便的字符串组合方式, 使用 `QString::arg()` 函数,

```
QString str;
str = QString("%1 was born in %2.")
    .arg("John")
    .arg(1982);
// str: "John was born in 1982."
```

例子中, “%1” 被替换为 “John”, “%2” 被替换为 “1982”。

函数 `QString::arg()` 的重载可以处理很多的数据类型。此外, 一些重载具有额外的参数对字段的宽度、数字基数或者浮点数精度进行控制。通常, 相对于 `QString::sprintf()`, `QString::arg()` 是一个比较好的解决方案, 因为它是类型安全的, 完全支持 Unicode, 并且允许改变 “%n” 参数的顺序。

`QString` 也提供了一些其他的组合字符串的方法, 包括:

- `insert()`, 在原字符串特定的位置插入另一个字符串;
- `prepend()`, 在原字符串的开头插入另一个字符串;
- `replace()` 函数, 用指定的字符串代替原字符串中的某些字符, 等等。

很多时候去掉一个字符串两端的空白 (空白字符包括回车字符 “\n”, 换行字符 “\r”, 制表符 “\t” 和空格字符 “ ” 等) 非常有用的, 比如获取用户输入的账号时。函数 `QString::trimmed()` 移除字符串两端的空白字符; 函数 `QString::simplified()` 除了移除字符串两端的空白字符外, 还会使用单个空格字符 “ ” 代替字符串中出现的空白字符。例如,

```
QString str = " How \t are \n you! ";
str = str.trimmed(); // str: "How \t are \n you! "
```

而如果使用 `str = str.simplified()`, 导致 `str` 的结果是 “How are you!”。

13.2.4 查询字符串数据

函数 `QString::startsWith()` 判断一个字符串是否以某个字符串开头, 例如,

```
QString str = "How ar you! Fine, thank you.";
str.startsWith("How", Qt::CaseSensitive); // 返回 true;
str.startsWith("Fine", Qt::CaseSensitive); // 返回 false;
```

函数 `QString::startsWith()` 具有两个参数, 第一个参数指定了一个字符串; 第二个参数指定是否大小写敏感 (默认情况下, 是大小写敏感的)。



类似的, `QString::endsWith()` 函数判断一个字符串是否以某个字符串结尾。

函数 `QString::contains()` 判断一个指定的字符串是否出现过, 例如,

```
QString str = "How ar you! Fine, thank you.";
str.contains("How", Qt::CaseSensitive); // 返回 true;
```

比较两个字符串也是经常使用的功能, `QString` 提供了多种比较手段:

- `operator <(const QString&)`, 比较一个字符串是否小于另一个字符串, 如果是返回 `true`;
- `operator <=(const QString&)`, 比较一个字符串是否小于等于另一个字符串, 如果是返回 `true`;
- `operator ==(const QString&)`, 比较两个字符串是否相等, 如果相等返回 `true`;
- `operator >=(const QString&)`, 比较一个字符串是否大于等于另一个字符串, 如果是返回 `true`;
- `localeAwareCompare(const QString&, const QString&)`, 静态函数, 比较前后两个字符串, 如果小于则返回负整数值; 如果等于返回 0; 如果大于返回正整数值。该函数的比较是基于本地 (locale) 字符集的, 而且是平台相关的。通常该函数用于向用户显示一个有序的字符串列表。
- `compare(const QString&, const QString&, Qt::CaseSensitivity)`, 该函数可以指定是否进行大小写的比较, 而大小写的比较是完全基于字符的 Unicode 编码值的, 而且是非常快的。返回值同 `localeAwareCompare()` 函数有些类似。

13.2.5 字符串的转换

`QString` 类提供了丰富的转换函数, 可以将一个字符串转换为数值类型或者转换为其他的字符编码集。

```
QString str = "125";
bool ok;
int hex = str.toInt(&ok, 16); // ok: true, hex: 293;
int dec = str.toInt(&ok, 10); // ok: true, dec: 125;
```

初始化一个 “125” 的字符串, 然后调用 `QString::toInt()` 函数将字符串转换为整形数值。函数 `QString::toInt()` 有两个参数, 第一个参数是一个 `bool` 类型的指针, 用于返回转换的状态: 当转换成功时设置为 `true`, 否则设置为 `false`; 第二个参数指定了转换的基数。

注意, 当基数设置为 0 时, 将会使用 C 语言的转换方法: 如果字符串以 “0x” 开头, 则基数为 10; 如果字符串以 “0” 开头, 则基数为 8; 其他情况下, 基数一律是 10。

除了刚才使用的 `QString::toInt()` 转换函数以外, `QString` 提供的数值转换函数还有 `toDouble()`、`toFloat()`、`toLong()`、`toLongLong()` 等。在此不再赘述。

`QString` 提供的字符编码集的转换函数有:

- `toAscii()`, 返回一个 ASCII 编码的 8 比特字符串;
- `toLatin1()`, 返回一个 Latin-1 (ISO 8859-1) 编码的 8 比特字符串;
- `toUtf8()`, 返回一个 UTF-8 编码的 8 比特字符串 (UTF-8 是 ASCII 码的超级, 它支持整个 Unicode 字符集);
- `toLocal8Bit()`, 返回一个系统本地 (locale) 编码的 8 比特字符串。

上面列出的转换函数, 将会返回一个 `const char*` 类型版本的 `QByteArray`, 即构造函数 `QByteArray(const char*)` 构造的 `QByteArray` 对象。`QByteArray` 类具有一个字节数组, 它既可以存储原始字节 (raw bytes), 也可以存储传统的以 “\0” 结尾的 8 比特字符串。在 Qt 中, 使用 `QByteArray` 比使用 `const char*` 更方便, 而 `QByteArray` 也支持隐式共享。

```

QString str = "How are you!";
QByteArray ba = str.toAscii();
qDebug() << ba;
ba.append("Fine, thank you.");
qDebug() << ba.data();

```

首先初始化一个字符串对象，然后通过 `QString::toAscii()` 函数，将 Unicode 编码的字符串转换为 ASCII 码的字符串，并存储在 `QByteArray` 对象 `ba` 中。接着使用 `qDebug()` 函数输出转换后的字符串（`qDebug()` 支持输出 Qt 对象）。最后，使用 `QByteArray::append()` 函数追加一个字符串并输出。

另外，需要注意的一个问题是，NULL 字符串和空（empty）字符串的区别。

一个 NULL 字符串就是使用 `QString` 的默认构造函数或者使用 “`(const char*)0`” 作为参数的构造函数创建的 `QString` 字符串对象；而一个空字符串是一个大小为 0 的字符串。一个 NULL 字符串一定是一个空字符串，而一个空字符串未必是一个 NULL 字符串。例如：

```

QString().isNull();      // 结果为 true;
QString().isEmpty();     // 结果为 true;
QString("").isNull();    // 结果为 false;
QString("").isEmpty();   // 结果为 true;

```

13.3 QVariant

`QVariant` 类看上去有点类似于 C++ 的联合（union）数据类型，它能够保存很多 Qt 类型的值，包括 `QColor`、`QBrush`、`QFont`、`QPen`、`QRect`、`QString`、`QSize` 等，也能够存放 Qt 的容器类型的值。Qt 很多的功能都是建立在 `QVariant` 的基础上的，比如 Qt 的对象属性以及数据库功能等。

下面，看一下 `QVariant` 类的应用。

```

// chapter13/variant/exam1.
#include <QDebug>
#include <QVariant>
#include <QColor>

int main(int argc, char *argv[])
{
    QVariant v(709);
    qDebug() << v.toInt();
}

```

声明一个 `QVariant` 变量 `v`，并初始化为一个整数。此时，`QVariant` 变量 `v` 包含了一个整数变量。接着，调用 `QVariant::toInt()` 函数将 `QVariant` 变量包含的内容转化为整数并输出。

```

v = QVariant("How are you!");
qDebug() << v.toString();

```

改变 `QVariant` 变量 `v` 包含的内容为一个字符串，并调用 `QVariant::toString()` 函数转化为要输出的字符串。

```

QMap<QString, QVariant> map;
map["int"] = 709;
map["double"] = 709.709;
map["string"] = "How are you!";

```



```
map["color"] = QColor(255, 0, 0);
QDebug() << map["int"] << map["int"].toInt();
QDebug() << map["double"] << map["double"].toDouble();
QDebug() << map["string"] << map["string"].toString();
QDebug() << map["color"] << map["color"].value<QColor>();
```

声明一个 QMap 变量 map，使用字符串作为键，QVariant 变量作为值。依次输入整数型、浮点型、字符串和 QColor 类型的值，并调用相应的转化函数并输出。

由于 QVariant 是 QtCore 模块的类，因此它没有为 QtGui 模块中的数据类型（例如，QColor、QImage 以及 QPixmap 等）提供转化函数，因此需要使用 QVariant::value() 函数或者使用 qVariantValue() 模板函数。在上面，在 QVariant 变量中保存了一个 QColor 对象，并使用模板 QVariant::value() 还原为 QColor，然后输出。

```
QStringList sl;
sl << "A" << "B" << "C" << "D";
QVariant slv(slv);
if(slv.type() == QVariant::StringList)
{
    QStringList list = slv.toStringList();
    for(int i=0; i<list.size(); ++i)
        qDebug() << list.at(i);
}
return 0;
}
```

创建了一个字符串列表，并将该列表保存在一个 QVariant 变量中。最后，输出列表的内容。

QVariant::type() 函数返回存储在 QVariant 变量中的值的数据类型。QVariant::StringList 是 Qt 定义的一个 QVariant::Type 枚举类型的变量，其他常用的类型枚举变量如表 13-7 所示。

表 13-7 Qt 常用类型枚举变量

变 量	对应的类型	变 量	对应的类型
QVariant::Invalid	无效类型	QVariant::Time	QTime
QVariant::Region	QRegion	QVariant::Line	QLine
QVariant::Bitmap	QBitmap	QVariant::Palette	QPalette
QVariant::Bool	bool	QVariant::List	QList
QVariant::Brush	QBrush	QVariant::SizePolicy	QSizePolicy
QVariant::Size	QSize	QVariant::String	QString
QVariant::Char	QChar	QVariant::Map	QMap
QVariant::Color	QColor	QVariant::StringList	QStringList
QVariant::Cursor	QCursor	QVariant::Point	QPoint
QVariant::Date	QDate	QVariant::Pen	QPen
QVariant::DateTime	QDateTime	QVariant::Pixmap	QPixmap
QVariant::Double	double	QVariant::Rect	QRect
QVariant::Font	QFont	QVariant::Image	QImage
QVariant::Icon	QIcon	QVariant::UserType	用户自定义类型

上述程序输出的结果如下：

```

709
"How are you!"
QVariant(int, 709) 709
QVariant(double, 709.709) 709.709
QVariant(QString, "How are you!") "How are you!"
QVariant(QColor, QColor(ARGB 1, 1, 0, 0) ) QColor(ARGB 1, 1, 0, 0)
"A"
"B"
"C"
"D"

```

13.4 Qt 的算法

Qt 的<QtAlgorithms>和<QtGlobal>模块提供了一些算法和函数。这一小节将简要地介绍一下几个经常使用的算法，更多的算法可以参考 Qt 的帮助文档。

```

#include <QDebug>

int main(int argc, char *argv[])
{
    double a = -19.3,
           b = 9.7;
    double c = qAbs(a);
    double max = qMax(b, c);
    int bn = qRound(b);
    int cn = qRound(c);
    qDebug() << "a = " << a;
    qDebug() << "b = " << b;
    qDebug() << "c = qAbs(a) = " << c;
    qDebug() << "qMax(b, c) = " << max;
    qDebug() << "bn = qRound(b) = " << bn << "cn = qRound(c) = " << cn;

    qSwap(bn, cn);
    qDebug() << "qSwap(bn, cn): " << "bn = " << bn << "cn = " << cn;

    return 0;
}

```

函数 qAbs() 返回 double 型数值 a 的绝对值，并赋值给 c。

函数 qMax()，返回两个数值的最大值。

函数 qRound() 返回一个浮点数的最接近的整数值，即四舍五入的返回一个整数值。

最后，调用 qDebug() 函数输出所有的计算结果。

编译运行上述程序，输出结果如下：

```

a = -19.3
b = 9.7
c = qAbs(a) = 19.3
qMax(b, c) = 19.3
bn = qRound(b) = 10 cn = qRound(c) = 19
qSwap(bn, cn): bn = 19 cn = 10

```



13.5 正则表达式

13.5.1 基本的正则表达式

使用正则表达式可以方便地完成处理字符串的一些操作，如验证、查找、替换、分割等。Qt 的 `QRegExp` 类是正则表达式的表示类，它基于 Perl 的正则表达式语言，完全支持 Unicode。

正则表达式由表达式 (expressions)、量词 (quantifiers) 和断言 (assertions) 组成。最简单的表达式是一个字符，要表示字符集的表达式可以使用类似如 “[AEIOU]”，表示匹配所有的大写元音字母。使用 “[^AEIOU]” 则表示匹配所有非元音字母，即辅音字母。连续的字符集使用可以使用表达式如 “[a-z]”，表示匹配所有小写英文字母。量词说明表达式出现的次数，例如 “x{1,2}” 表示 “x” 可以至少有一个，至多两个。

在计算机语言中，标识符通常要求以字母或下划线开头，后面可以是字母、数字和下划线。则满足条件的标识符表示为 “[A-Za-z_]+[A-Za-z_0-9]*”。

表达式中的 “+” 表示 “[A-Za-z_]” 至少出现一次，可以出现多次。“*” 表示 “[A-Za-z_0-9]” 可以出现零次或多次。类似的量词如表 13-8 所示。

表 13-8 正则表达式的量词

量 词	含 义
$E?$	匹配 0 次或 1 次
E^+	匹配 1 次或多次
E^*	匹配 0 次或多次
$E\{n\}$	匹配 n 次
$E\{n,\}$	至少匹配 n 次
$E\{,m\}$	最多匹配 m 次
$E\{n,m\}$	至少匹配 n 次，最多匹配 m 次

如下的程序打印出了 C++ 源程序中的所有标识符。

```
#include <iostream>
#include <QtCore>

using namespace std;

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    QStringList capList;
    int pos;
    QRegExp rx("[A-Za-z_]+[A-Za-z_0-9]*");

    QFile file("main.cpp");
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
        return -1;
```

```

QTextStream in(&file);
while (!in.atEnd()) {
    QString str = in.readLine();
    pos = 0;
    while ({pos = rx.indexIn(str, pos)} != -1) {
        capList = rx.capturedTexts();
        cout << qPrintable(capList.at(0)) << "\t";
        pos += rx.matchedLength();
    }
    cout << endl;
}
return 0;
}

```

由于是控制台程序，所以需在 `qmake` 文件中加入：

```

QT = core
CONFIG += console

```

在正则表达式中，有一些字符有特殊的含义，表 13-9 给出了一些常用的转义字符：

表 13-9 正则表达式常用转义字符

转义字符	说 明
\n	ASCII 换行字符 (LF, 0x0A)
\r	ASCII 回车字符 (CR, 0x0D)
\t	ASCII 水平制表符 (HT, 0x09)
\xhhhh	Unicode 的十六进制代码 (0x0000-0xFFFF)
\Oooo	表示八进制的 ASCII/Latin 字符 (0-0377, 即十进制的 0-255)
.	匹配任意字符
\d	匹配一个数字
\D	匹配非数字
\s	匹配空格
\S	匹配非空格
\w	匹配单词字符
\W	匹配非单词字符
\n	在表达式内第 <i>n</i> 个捕获的子表达式文本

“\n”用来在正则表达式中引用已经捕获的文字，下节将介绍其具体用法。需要注意的是由于在 C++ 中“\”也是转义字符，所以要在 C++ 中使用正则表达式的“\”，必须再转义一次。如果要使用“\w”，则应该写为“\\w”，如果要使用“\”本身，则使用“\\”。如果不想自己转义，可以使用 `QRegExp` 的 `escape()` 函数进行转义。

要在正则表达式中匹配一个单词可以使用“\bzeki\b”，其中“\b”表示单词边界，即非单词字符，如空格、换行和字符串开始或结束。如果要在字符串开头匹配一个单词，使用“^zeki”形式，在字符串结尾匹配则使用“zeki\$”的形式。“^”、“\$”、“\b”都是正则表达式的断言，正则表达式断言如表 13-10 所示。



表 13-10 正则表达式新言

符 号	含 义
^	表示在字符串开头进行匹配
\$	表示在字符串结尾进行匹配
\b	单词边界
\B	非单词边界
(?=E)	表示表达式后紧随 E 才匹配
(?E)	表示表达式后不跟随 E 才匹配

例如要在 using 后面是 namespace 时才匹配 using，可以使用“using(=?s+namespace)”。这样匹配的结果是 using，如果使用正则表达式“using\s+namespace”则匹配为 using namespace。如果使用表达式“using(?E\s+namespace)”则表示只有在 using 后面不是 namespace 时才匹配 using。

13.5.2 文字捕获

在正则表达式中，可以使用小括号将特定表达式括起来，以捕获特定的文字。如下的代码展示了捕获文字中的“海里(nm)”或“公里(km)”。

```
QRegExp rx("(\\d+)(?:\\s*)(km|nm)");
int pos = rx.indexIn("Distance: 3240 nm");
if (pos > -1) {
    qDebug() << rx.cap(1) << rx.cap(2);
}
```

在上面的代码中，cap(1)返回“3240”，cap(2)返回“nm”。QRegExp 使用 cap()和 capturedTexts()函数来返回捕获的文字。cap(0)返回所有捕获的文字，cap(n)返回第 n 个捕获的文字。capturedText()返回一个字符串列表，第 0 个元素是整个捕获的文字。从第一个元素开始，就是捕获的第 n 个子串。对于上面的代码，capturedTexts()的第 0 个元素是“3240 nm”，然后是“3240”、“nm”。

如果一个子表达式只希望用括号括起来，而不希望成为被捕获的子串时，可以使用“?;”，如上面的“(?:\\s*)”就没有被当成要捕获的子串。

下面使用一个简单的例子来捕获 HTML 文件中的 http、ftp、mailto 的网址。

```
#include <iostream>
#include <QtCore>

using namespace std;

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    int pos;
    int index;
    QRegExp rx("(http://(?:[a-z0-9_\\.]+(?:com|cn|net|org|gov|mil)|"
        "(ftp://(?:[a-z0-9_\\.]+(?:com|cn|net|org|gov|mil)|"
        "(mailto:(?:[a-z0-9_\\.]+@(?:[a-z0-9_\\.]+)"
        "+(?:com|cn|net|org|gov|mil))))))");
    if (!rx.isValid())
```



```

{
    cout << qPrintable(rx.errorString());
    return -1;
}

QFile file("index.html");
if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
    return -1;
QTextStream in(&file);
while (!in.atEnd()) {
    QString str = in.readLine();
    pos = 0;
    while ((pos = rx.indexIn(str, pos)) != -1) {
        for(index = 0; index <3; index++) {
            if(!rx.cap(index).isEmpty())
                cout << qPrintable(rx.cap(index)) << endl;
        }
        pos += rx.matchedLength();
    }
}
return 0;
}

```

在程序中使用了 `isValid()` 函数来检测正则表达式是否有效，如果无效则打印出错误信息 `errorString()`。

Qt 的正则表达式有 4 种语法，默认的语法为 `QRegExp::RegExp`。`QRegExp` 的默认语法是 `QRegExp::RegExp`，是类似于 Perl 的语法，不使用贪婪匹配。`QRegExp::RegExp2` 是一种贪婪匹配的语法，即尽可能多地匹配。例如 “`x*(x)*`” 正则表达式匹配 `xxxx` 将是 `xxxx`，而不是 `x`。在 Qt 5 中 `RegExp2` 将成为默认的语法。第三种是通配符，如同在 shell 中使用的语法一样，用 “`?`” 代表任意单个字符，“`*`” 表示 0 个或多个字符，“`[...]`” 表示字符集。最后一种是 `QRegExp::FixedString`，就是字符原义，没有任何特殊转义。

如果要使用通配符可以使用如下的语法。

```

QRegExp rx(*.cpp);
rx.setPatternSyntax(QRegExp::Wildcard);
rx.exactMatch(*.cpp);

```

使用通配符时，要使用 `exactMatch()`，相当于字符串首尾都必须匹配。例如上面的代码中 `main.cpp~` 就不能匹配，因为该文件名以 `cpp~` 结尾。

理解正则表达式的办法是多实践，在 Qt 的例子中有一个正则表达式的例子，在 `examples/tools/regex` 目录下，可以测试各种正则表达式。

13.6 小 结

在这一章，学习了 Qt 提供的一些常用模板和工具类，包括 `QString` 类、`QVariant` 类以及 Qt 的容器类和正则表达式等，也附带介绍了隐式共享技术。但是同 Qt 的整个模板库和工具类相比，这只是冰山一角。



高级篇

第 14 章 XML

第 15 章 模型/视图结构

第 16 章 高级绘图

第 17 章 进程与进程间通信

第 18 章 Qt 插件

第 19 章 脚本——QtScript

第 20 章 国际化

第 21 章 Qt 单元测试框架

第 14 章 XML

Qt 的 XML 模块支持流行的两种 XML 解析方法，DOM 和 SAX。两种方法各有优缺点，DOM 方法将 XML 文件表示成一棵树，便于随机访问其中的节点，但消耗内存相对多一些。SAX 是一种事件驱动的 XML API，速度快，但不便于随机访问任意节点。通常根据实际应用选用合适的解析方法。在 Qt 4.3 中，还引入了一种基于流的 XML 解析方法。

为了在 Qt 程序中使用 XML API，必须在 qmake 工程文件中加入如下一行：

```
QT += xml
```

要使用 Qt 的 QtXml 模块，可以只包含具体 XML 类的头文件，也可以简单地加入

```
#include <QtXml>
```

包含所有的 Qt XML 头文件。

14.1 DOM

14.1.1 DOM 入门

DOM（文档对象模型，Document Object Model）是 W3C 开发的独立于平台和语言的接口，它可以使程序和脚本能够动态地存取和更新 XML 文档的内容、结构和风格。DOM 有 Level 1、2 和 3 三个级别的规范。Qt 支持 DOM Level 2 规范。

DOM 在内存中将 XML 文件表示为一棵树，用户通过 API 可以随意地访问树的任意节点内容。在 Qt 中，XML 文档自身使用 QDomDocument 表示，所有的节点类都从 QDomNode 继承。其相关类的关系如图 14-1 所示。

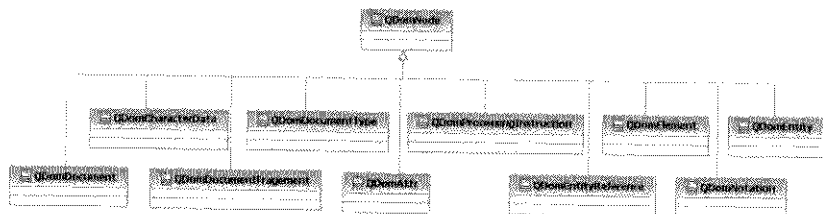


图 14-1 Qt DOM 相关类的关系

14.1.2 使用 DOM

在 DOM API 中，使用 QDomDocument 类表示一个 XML 文档。可以按如下方式读取 XML 文档。

```

QDomDocument doc("test");
QFile file("test.xml");
if (!file.open(QIODevice::ReadOnly))
    return;
if (!doc.setContent(&file)) {
    file.close();
    return;
}
file.close();

```

成员函数 `QDomDocument::setContent()` 完成 XML 文档的设置，它从 `QFile` 对象中读取 XML 数据并检测 XML 文档的编码。`setContent()` 有几种重载形式，可以分别从 `QByteArray`、`QString`、`QIODevice`、`QXmlInputSource` 中读取 XML 数据。

可以读出 XML 文件第一层的所有元素并显示。

```

QDomElement docElem = doc.documentElement();
QDomNode n = docElem.firstChild();
while(!n.isNull()) {
    QDomElement e = n.toElement();
    if(!e.isNull()) {
        qDebug() << e.tagName() ;
    }
    n = n.nextSibling();
}

```

通过函数 `documentElement()` 取出 XML 文件中的根元素，然后使用 `firstChild()` 得到根元素的第一个节点，使用循环逐步将所有顶层元素的标签名输出。函数 `nextSibling()` 取得一个节点的下一个兄弟节点。

下面实现一个读取 SVG 文件中的部分绘图元素并绘制其图形的例子来学习 DOM 的应用。SVG 文件是用 XML 表示的矢量图形文件，每种图形都使用 XML 标签表示。例如在 SVG 中画折线的标签如下：

```
<polyline fill="none" stroke="#888888" stroke-width="2" points="100,200 100,100" />
```

其中 `polyline` 表示绘制折线，`fill` 属性表示填充，`stroke` 表示画笔颜色，`stroke-width` 表示画笔宽度，`points` 表示折线的点。

在例子中，为了不使程序太复杂，仅考虑只包含绘制折线、圆、矩形等简单图形的 SVG 标签，而且只处理部分属性。读取 SVG 文件的函数实现如下：

```

void DrawWidget::readSvg(QString fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly))
        return;
    if (!doc.setContent(&file)) {
        file.close();
        return;
    }
}

```



```
file.close();  
bDraw = true;  
update();  
}
```

可以看到，只需打开文件并使用 `setContent()` 函数就完成了 XML 文件 DOM 树的初始化。
绘制过程如下：

```
void DrawWidget::paintEvent(QPaintEvent *event)  
{  
    if(!bDraw)  
        return;  
    QPainter painter(this);  
    QDomElement root = doc.documentElement();  
    QDomElement child = root.firstChildElement();  
    QRect rect;  
    while (!child.isNull()) {  
        if(child.tagName() == "rect") {  
            setPenAndBrush(painter, child);  
            rect.setRect(child.attribute("x").toInt(),  
                child.attribute("y").toInt(),  
                child.attribute("width").toInt(),  
                child.attribute("height").toInt());  
            painter.drawRect(rect);  
        }  
        else if (child.tagName() == "circle") {  
            setPenAndBrush(painter, child);  
            int cx=child.attribute("cx").toInt();  
            int cy=child.attribute("cy").toInt();  
            int r=child.attribute("r").toInt();  
            rect.setRect(cx-r, cy-r, 2*r, 2*r);  
            painter.drawEllipse(rect);  
        }  
        else if (child.tagName() == "text") {  
            setPenAndBrush(painter, child);  
            int x=child.attribute("x").toInt();  
            int y=child.attribute("y").toInt();  
            QFont font;  
            font.setFamily(child.attribute("font-family"));  
            font.setPointSize(child.attribute("font-size").toInt());  
            painter.setFont(font);  
            painter.drawText(x, y, child.text());  
        }  
        else if (child.tagName() == "polyline") {  
            setPenAndBrush(painter, child);  
            QString points = child.attribute("points");  
            QStringList list = points.split(" ");  
            QPoint pointArray[100];
```

```

int i=0;
foreach(QString point, list) {
    int index = point.indexOf(",");
    pointArray[i].setX(point.left(index).toInt());
    pointArray[i].setY(point.mid(index + 1).toInt());
    i++;
}
painter.drawPolyline(pointArray, i);
}
child = child.nextSiblingElement();
}
}

```

绘制函数根据 `tagName()` 返回的标签名分类处理, 使用 `attribute()` 函数读出元素的属性, 并设置相应的画笔和画刷, 设置画笔和画刷的函数实现如下:

```

void DrawWidget::setPenAndBrush(QPainter& painter, QDomElement& dom Element)
{
    bool ok;
    QPen pen;
    QBrush brush;
    QString fill = domElement.attribute("fill");
    if(fill != "none") {
        QRgb color = (QRgb)fill.toInt(&ok, 16);
        brush.setColor(QColor(color));
    }
    QString stroke = domElement.attribute("stroke");
    if(stroke != "none") {
        if(stroke == "blue")
            pen.setColor(Qt::blue);
        else if(stroke == "red")
            pen.setColor(Qt::red);
        else {
            QRgb color = (QRgb)stroke.toInt(&ok, 16);
            pen.setColor(QColor(color));
        }
    }
    QString strokeWidth = domElement.attribute("stroke-width");
    pen.setWidth(strokeWidth.toInt());
    painter.setPen(pen);
    painter.setBrush(brush);
}

```

在这里, 通过 `tagName()` 获取标签名, 确定要绘制的图形。使用 `attribute()` 函数获取画笔、画刷属性和图形位置。使用 `nextSiblingElement()` 获取下一个标签。

程序运行绘出的图形如图 14-2 所示。

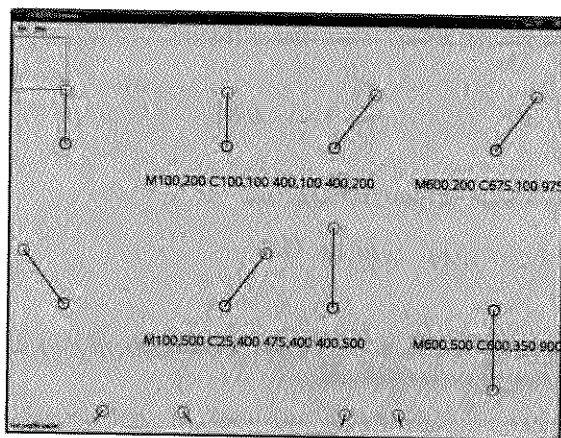


图 14-2 DOM 应用

14.1.3 使用 DOM 写 XML 文件

使用 DOM 创建一个 XML 文件是相当简单的。首先创建一个 `QDomDocument` 对象，接着创建相应的节点和属性，并建立相应的父子关系，最终形成一棵 DOM 树。但是 `QDomDocument` 本身并不具备写入 XML 文件的能力，需要将 DOM 树和 `QTextStream` 对象关联，写入磁盘文件。

很多开发工具的工程文件都用 XML 文件表示，KDevelop 也不例外。下面就用 Qt 的 DOM 类来创建一个简单的 KDevelop 工程文件。

```
#include <iostream>
#include <QtXml>

using namespace std;

int main(int argc, char *argv[])
{
    QFile file("domwrite.kdevelop");
    if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate))
    {
        cout << "Can't create file!" << endl;
        return 1;
    }
    QDomDocument doc;
    QDomText text;
    QDomElement element;
    QDomProcessingInstruction instruction;
    instruction = doc.createProcessingInstruction
        ("xml", "version='1.0'");
    doc.appendChild(instruction);
```



```
QDomElement root = doc.createElement("kdevelop");
doc.appendChild(root);
QDomElement general = doc.createElement("general");
root.appendChild(general);

element = doc.createElement("author");
text = doc.createTextNode("zeki");
element.appendChild(text);
general.appendChild(element);

element = doc.createElement("email");
text = doc.createTextNode("caizhiming@tom.com");
element.appendChild(text);
general.appendChild(element);

element = doc.createElement("version");
text = doc.createTextNode("$VERSION");
element.appendChild(text);
general.appendChild(element);

element = doc.createElement("projectmanagement");
text = doc.createTextNode("KdevTrollProject");
element.appendChild(text);
general.appendChild(element);

element = doc.createElement("primarylanguage");
text = doc.createTextNode("C++");
element.appendChild(text);
general.appendChild(element);

QDomElement keywords = doc.createElement("keywords");
element = doc.createElement("keyword");
text = doc.createTextNode("C++");
element.appendChild(text);
keywords.appendChild(element);
general.appendChild(keywords);

element = doc.createElement("projectname");
text = doc.createTextNode("domwrite");
element.appendChild(text);
general.appendChild(element);

element = doc.createElement("ignoreparts");
general.appendChild(element);

element = doc.createElement("projectdirectory");
text = doc.createTextNode(".");
element.appendChild(text);
```



```
general.appendChild(element);

element = doc.createElement("absoluteprojectpath");
text = doc.createTextNode("false");
element.appendChild(text);
general.appendChild(element);

element = doc.createElement("description");
general.appendChild(element);

element = doc.createElement("defaultencoding");
general.appendChild(element);

QDomElement kdevfileview = doc.createElement("kdevfileview");
QDomElement groups = doc.createElement("groups");
element = doc.createElement("group");
QDomAttr pattern = doc.createAttribute("pattern");
pattern.setValue("*.cpp;*.cxx;*.h");
QDomAttr name = doc.createAttribute("name");
name.setValue("Sources");
element.setAttributeNode(pattern);
element.setAttributeNode(name);
groups.appendChild(element);
kdevfileview.appendChild(groups);
root.appendChild(kdevfileview);

QTextStream out(&file);
doc.save(out, 4);

return 0;
}
```

从上面的例子可以看出,创建 XML 文件必须先使用 `QFile` 创建实在的文件,再通过 `QTextStream` 和 `QDomDocument` 类关联,这样才可以使用 `QDomDocument` 的 `save()` 方法来保存 DOM 树的内容。

创建不同的节点使用不同的方法。创建 XML 处理指令使用 `createProcessing Instruction()`, 创建标签使用 `createElement()`, 创建属性使用 `createAttribute()` 方法。将各种节点加入到 DOM 树的方法也不尽相同。`appendChild()` 添加一个标签, `setAttributeNode()` 添加属性, `replaceChild()` 替换节点, `removeChild()` 删除节点, `insertBefore()` 在指定节点之前插入节点, `insertAfter()` 在指定节点之后插入节点。

14.2 SAX

DOM 方法需要读取整个文件并将它存储到树结构中,因而效率不高、缓慢,并且可能会过度使用资源。一种替代方法是使用 SAX (Simple API for XML)。SAX 允许在读取文档时处理该文档,这避免了在采取操作之前等待存储文档的所有内容。

SAX 是由 XML-DEV 邮件列表的成员开发的。他们的目的是提供一种更自然的方法来使用 XML,这种方法不会涉及使用 DOM 的那种开销。SAX 是基于事件的 API, SAX 解析器将事件(如元素的开

始或结束) 发送给处理信息的事件处理程序。然后应用程序自己可以处理数据。虽然原始文档保持不变, 但 SAX 提供了操作数据的方法, 然后将该方法导向另一个过程或文档。

SAX 没有官方的标准。W3C 或其他官方组织不维护 SAX, 但在 XML 社区中, 它是一个实际上的标准。SAX 的官方网站是: “<http://www.saxproject.org>”。

SAX2 是一种“推进模型”分析器。程序员提供处理程序, 在发生特定事件时(例如开始一个文档, 或者一个元素的开始和结束), 分析器便调用这些事件处理程序。SAX2 分析器生成这些事件, 包括在 XML 文档内容中发生的事件、在 DTD 中发生的事件和错误事件。应用程序只需要为感兴趣的事件实现处理程序。如果没有为特定类型的事件实现处理程序, 那么该事件将被忽略。

到目前为止, 有两个 SAX 版本: SAX1 和 SAX2。Qt 4 不支持 Java 风格的 SAX1 API, 只支持 SAX2 API。SAX1 与 SAX2 很相似, 只是它缺少名称空间处理。

建立一个 SAX 分析程序的过程是:

- 01 创建 SAXXMLReader 实例;
- 02 设置处理程序类;
- 03 设置 XML 文档的来源;
- 04 开始分析。

在分析器读取 XML 文档时, 基于事件的语法分析器将事件发送给应用程序。这些事件类似于用户界面事件, 例如, Qt 中的各种用户界面事件和信号。事件通知应用程序发生了某个事件并需要应用程序做出反应。

在 XML 语法分析器中, 事件与用户操作无关, 而与正在读取的 XML 文档中的元素有关。有关于以下方面的事件:

- 元素开始和结束标记;
- 元素内容;
- 实体;
- 语法分析错误。

在 Qt 中, 对 XML、DTD 等都有相应的事件处理类:

- QXmlContentHandler 类产生 XML 文档的事件(如开始一个标签或字符);
- QXmlDTDHandler 类产生 DTD 对应的事件;
- QXmlErrorHandler 类产生分析错误时的事件;
- QXmlEntityResolver 类产生允许用户解析外部实体的事件;
- QXmlDeclHandler 类产生更多的 DTD 相关的事件;
- QXmlLexicalHandler 类报告文档的词法结构事件。

实现一个 QXmlContentHandler 通常要实现 startElement()、endElement()、startDocument()、endDocument()、characters() 等事件。

SAX 尽管非常灵活, 但是随机访问的能力不如 DOM, 所以在具体使用时应该按实际考虑。

在 Qt 中, ui 文件就是一种 XML 文件。uic 将 ui 文件转换为 C++ 的头文件。这里实现用 SAX 读取 ui 文件, 并生成简单的代码文件。为了不使程序太复杂, 只处理 ui 文件中的部分标签。

XML 的 Handler 类从 QXmlDefaultHandler 继承。

```
class UiHandler : public QXmlDefaultHandler
{
public:
```



```

UiHandler(QTextEdit *textEdit, QString fileName);

bool startElement(const QString &namespaceURI, const QString &localName,
                  const QString &qName, const QDomAttributes &attributes);
bool endElement(const QString &namespaceURI, const QString &localName,
                const QString &qName);
bool characters(const QString &str);
bool fatalError(const QDomParseException &exception);
QString errorString() const;

private:
    QString macroName;
    QString propName;
    QString currentText;
    QString topWidget;
    QString currentWidget;
    QString constructorText;
    QString errorStr;
    QString classAttr;
    QString nameAttr;
    QString xTag;
    QString yTag;
    QString widthTag;

    short level;
    QTextEdit *txtEdit;
    QTextCursor cursor;
};

```

SAX 的事件处理实现如下:

```

#include <QtGui>
#include "uihandler.h"

```

```

UiHandler::UiHandler(QTextEdit *textEdit, QString fileName)
{
    txtEdit = textEdit;
    macroName = fileName.toUpper() + "_H";
    level = 0;
}

```

构造函数中的 macroName 用来生成防止头文件重复包含的宏, 形式如下:

```

#ifndef UI_ABCDEFG_H
#define UI_ABCDEFG_H
...
#endif

```

XML 元素开始的响应函数如下:

```

bool UiHandler::startElement(const QString & /* namespaceURI */,
                             const QString & /* localName */,
                             const QString &qName,
                             const QDomAttributes &attributes)
{
    if (qName == "ui") {
        QString version = attributes.value("version");
        if (!version.isEmpty() && version != "4.0") {
            errorStr = QObject::tr("The file is not an Qt version 4.x file.");
            return false;
        }
    }
    else {
        txtEdit->clear();
        cursor = txtEdit->textCursor();
        cursor.insertText("// Generated by sax parser test.\n\n");
        cursor.insertText("#ifndef " + macroName + "\n");
        cursor.insertText("#define " + macroName + "\n\n");
    }
    else if (qName == "widget") {
        classAttr = attributes.value("class");
        nameAttr = attributes.value("name");
        level++;
        if (level == 1) {
            topWidget = nameAttr;
            cursor.insertText("class " + nameAttr + " : public " +
                             classAttr + "\n");
            cursor.insertText("{\n");
            cursor.insertText("public:\n");
            constructorText = "\n\t" + nameAttr + "()\n";
        }
        else {
            cursor.insertText("\t" + classAttr + " *" + nameAttr + ";\n");
            constructorText += "\t\t" + nameAttr + " = new " + classAttr + ";\n";
        }
    }
    else if (qName == "property") {
        propName = attributes.value("name");
    }

    currentText.clear();
    return true;
}

```

这里根据标签的不同，分别进行了处理。对于<ui>标签，判断 ui 文件的版本并生成相应的防止头文件包含的宏。对于<widget>标签，还要判断是否是顶层 Widget，以便生成特定的代码。对于<property>标签，则暂时记录下来属性的值。

元素结束时的响应函数如下：

```

bool UiHandler::endElement(const QString & /* namespaceURI */,
                           const QString & /* localName */,
                           const QString &qName)

```



```
{
    if (qName == "ui") {
        cursor.insertText("#endif\n");
    }
    else if (qName == "widget") {
        if (level == 1) {
            constructorText += "\t\n";
            cursor.insertText(constructorText);
            cursor.insertText("; \n");
        }
        level--;
    }
    else if (qName == "string") {
        if (propName == "windowTitle") {
            constructorText += "\t\tsetWindowTitle(tr(\"" + currentText
                + "\")); \n";
        }
        else if (propName == "text") {
            constructorText += "\t\t" + nameAttr + "->setText(tr(\""
                + currentText + "\")); \n";
        }
    }
}
else if (qName == "x") {
    if (propName == "geometry") {
        xTag = currentText;
    }
}
else if (qName == "y") {
    if (propName == "geometry") {
        yTag = currentText;
    }
}
else if (qName == "width") {
    if (propName == "geometry") {
        widthTag = currentText;
    }
}
else if (qName == "height") {
    if (propName == "geometry") {
        if (level == 1) {
            constructorText += "\t\tresize(" + widthTag + ", " + currentText
                + "); \n";
        }
        else {
            constructorText += "\t\t" + nameAttr +
                "->setGeometry(QRect(" + xTag + ", " + yTag + ", " +
                widthTag + ", " + currentText + ")); \n";
        }
    }
}
```

```

    }
    }
    return true;
}

```

元素结束处理的函数较元素开始的函数内容多一些，因为元素开始时，只需要记录一些值就可以了，但元素结束时，要进行真正的处理。这里对不同的标签，生成了不同的代码。

处理字符的响应函数相对要简单一些，只需要记录文字就可以了。

```

bool UiHandler::characters(const QString &str)
{
    currentText += str;
    return true;
}

```

最后是错误处理的函数，这里也只是简单地处理并返回相应的值就可以了。

```

bool UiHandler::fatalError(const QDomParseException &exception)
{
    return false;
}

```

```

QString UiHandler::errorString() const
{
    return errorStr;
}

```

程序解析的界面如图 14-3 所示。

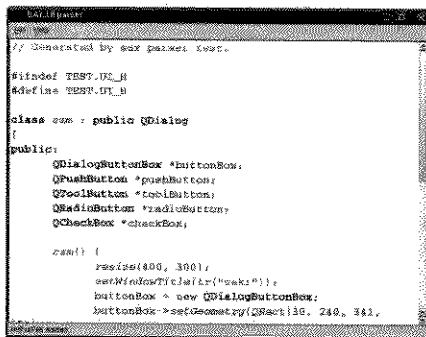


图 14-3 SAX 分析 ui 文件

14.3 基于流的 XML API

Qt 4.3 中引入了 `QXmlStreamReader` 和 `QXmlStreamWriter` 类。`QXmlStreamReader` 提供了一种快速、简单的 XML 流 API，它可以从 `QIODevice` 或 `QByteArray` 读取 XML 数据。使用 `QXmlStreamReader` 处理 XML 比使用 Qt 的 SAX XML 处理器更为自然，不需要像上一节的 SAX 程序一样记录大量状态。



QXmlStreamReader 类似于 SAX 分析器，也是在读取流的时候报告 XML 文档的符号，具有 SAX 分析器占用内存小的优点。不同于 SAX 分析器的是，在 SAX 中，必须提供处理 XML 事件的处理器，用户只是被动地响应事件；而在 QXmlStreamReader 中，用户控制了流程，并在需要处理符号的时候主动拉出 XML 符号。使用拉模式的 XML 解析的优点是用户可以构造递归的解析器，从而可以将 XML 解析代码根据处理要求分为不同的类或函数来实现，从而使代码更为清晰。同时也能够很容易地跟踪 XML 的解析状态。

在 QXmlStreamReader 中，通过 readNext() 读出需要的符号。使用 isProcessingInstruction(), isStartElement(), isEndElement(), isWhitespace(), isCharacters() 等函数判断读出的符号类型。符号类型用枚举型 QXmlStreamReader::TokenType 表示，具体如表 14-1 所示。

表 14-1 QXmlStreamReader 的符号值

类 型	说 明
QXmlStreamReader::NoToken	没有读到任何符号
QXmlStreamReader::Invalid	产生错误
QXmlStreamReader::StartDocument	文档开始
QXmlStreamReader::EndDocument	文档结束
QXmlStreamReader::StartElement	元素开始
QXmlStreamReader::EndElement	元素结束
QXmlStreamReader::Characters	读到字符
QXmlStreamReader::Comment	读到注释
QXmlStreamReader::DTD	DTD
QXmlStreamReader::EntityReference	实体引用
QXmlStreamReader::ProcessingInstruction	处理指令

一个典型的 QXmlStreamReader 的程序如下所示：

```
QXmlStreamReader xmlReader;
...
while (!xmlReader.atEnd()) {
    xmlReader.readNext();
    ... //进行相应的处理
}
if (xmlReader.hasError()) {
    ... // 错误处理
}
```

下面使用 QXmlStreamReader 实现上一节使用 SAX 的实现的例子。对 SAX 和 QXmlStreamReader 进行一个比较。主程序框架都是一样的，只需要把上一节的 UiHandler 类换为 UiReader 类即可，UiReader 类定义如下：

```
#ifndef UIREADER_H
#define UIREADER_H

#include <QtGui>
#include <QXmlStreamReader>
```



```

class UiReader : public QDomStreamReader
{
public:
    UiReader(QTextEdit *textEdit, QString fileName);
    bool read(QIODevice *device);

private:
    QString macroName;
    QString propName;
    QString classAttr;
    QString nameAttr;
    QString currentText;
    QString topWidget;
    QString currentWidget;
    QString constructorText;
    QString errorStr;
    QString xTag;
    QString yTag;
    QString wTag;
    QString hTag;
    short level; // 记录 Widget 的层次关系
    QTextEdit *textEdit;
    QTextCursor cursor;

    void readUi();
    void readWidget();
    void readProperty();
    void readRect();
};

#endif

```

类的成员变量和上一节类似。下面实现该类，构造函数和 SAX 的也是一致的。

```

#include <QtGui>
#include <QDebug>
#include "uireader.h"

UiReader::UiReader(QTextEdit *textEdit, QString fileName)
{
    textEdit = textEdit;
    macroName = fileName.toUpper() + "_H";
    level = 0;
}

```

读取 XML 流的函数如下：

```

bool UiReader::read(QIODevice *device)
{
    setDevice(device);
}

```

```

while(!atEnd()) {
    readNext();

    if(isStartElement()) {
        if(name() == "ui" && attributes().value("version") == "4.0")
            readUi();
        else
            raiseError(QObject::tr("The file is not an Qt 4.x UI file."));
    }
}

return !error();
}

```

在函数中判断文件是否为 Qt 4 的 ui 文件，如果不是则产生错误；否则调用 readUi() 读取 ui 文件。可以看到，主动权掌握在用户手里，而不是分析器手里。

readUi() 函数真正完成 ui 文件的读取，其实现如下。

```

void UiReader::readUi()
{
    txtEdit->clear();
    cursor = txtEdit->textCursor();
    cursor.insertText("// Generated by stream xml parser test.\n\n");
    cursor.insertText("#ifndef " + macroName + "\n");
    cursor.insertText("#define " + macroName + "\n\n");

    while(!atEnd()) {
        readNext();

        if(isStartElement()) {
            if(name() == "widget")
                readWidget();
        }
    }

    cursor.insertText("#endif\n");
}

```

在该函数中，又调用了 readWidget() 函数，其实现如下。

```

void UiReader::readWidget()
{
    classAttr = attributes().value("class").toString();
    nameAttr = attributes().value(QString("name")).toString();
    if(level == 0) {
        cursor.insertText("class " + nameAttr + " ; public " +
            classAttr + "\n");
        cursor.insertText("\n");
        cursor.insertText("public:\n");
        constructorText = "\n\t" + nameAttr + "{}\n";
    }
}

```

```

        constructorText = "\n\t" + nameAttr + "()" {\n"; // 构造函数
    }
    else {
        cursor.insertText("\t" + classAttr + " *" + nameAttr + ";\n");
        constructorText += "\t\t" + nameAttr + " = new " + classAttr + ";\n";
    }

    ++level;
    while(!atEnd()) {
        readNext();

        if(isEndElement() && name() == "widget")
            break;

        if(isStartElement()) {
            if(name() == "property") {
                readProperty();
            }
            else if(name() == "widget") {
                readWidget();
                --level;
            }
        }
    }

    if(level == 1) {
        constructorText += "\t}\n";
        cursor.insertText(constructorText);
        cursor.insertText(");\n");
    }
}

```

函数中还是使用 `level` 变量记录了 `widget` 标签的嵌套层数，对顶层的 `widget` 和第二层的处理方式不一样，需要读取 `widget` 属性时，调用 `readProperty()` 函数，其实现如下。

```

void UiReader::readProperty()
{
    propName = attributes().value(QString("name")).toString();
    while(!atEnd()) {
        readNext();

        if(isEndElement() && name() == "property")
            break;

        if(isStartElement()) {
            if(name() == "string") {
                readNext();
                if(propName == "windowTitle") {

```



```
        constructorText += "\\t\\tsetWindowTitle{tr(\"\" +  
            text().toString() + "\\");\\n";  
    }  
    else if(propName == "text") { // QComboBox is incorrect.  
        constructorText += "\\t\\t\" + nameAttr + \"->setText{tr(\"\" +  
            + text().toString() + "\\");\\n";  
    }  
    }  
    else if(name() == "rect") {  
        readRect();  
    }  
    }  
    }  
}
```

该函数读取属性，如果要读取 rect 标签，则调用 readRect()函数，其实现如下所示。

```
void UiReader::readRect()  
{  
    while(!atEnd()) {  
        readNext();  
  
        if(isEndElement() && name() == "rect")  
            break;  
  
        if(isStartElement()) {  
            if(name() == "x") {  
                xTag = readElementText();  
            }  
            else if(name() == "y") {  
                yTag = readElementText();  
            }  
            else if(name() == "width") {  
                wTag = readElementText();  
            }  
            else if(name() == "height") {  
                hTag = readElementText();  
            }  
        }  
    }  
    if(level == 1) {  
        constructorText += "\\t\\tresize(\" + wTag + \", \" + hTag + \");\\n";  
    }  
    else {  
        constructorText += "\\t\\t\" + nameAttr + \"->setGeometry(QRect(\" +  
            xTag + \", \" + yTag + \", \" + wTag + \", \" + hTag + \");\\n";  
    }  
}
```

从上面的代码可以看出，虽然总的代码行数比 SAX 程序增加了，但是由于各个标签能够独立使用一个函数进行处理，显得层次分明。

14.4 小 结

Qt 支持 DOM 和 SAX 的两种 XML 文件解析方式。DOM 形成的树随机访问能力强，但占资源多一些；SAX 是事件驱动的，占用资源少，但 XML 没有 DOM 灵活。具体使用哪种方式看应用的类型。在 Qt 4.3 中还引入了一种基于流的解析方式，相对于 SAX，这种方式能更自然地处理 XML 文件，而且使用户掌握了主动权。

第 15 章 模型/视图结构

Qt 4 中引入了模型/视图框架来完成数据与表现的分离,这在 Qt 4 中称为 InterView 框架。通过 InterView,可以将一种模型以不同的视图表现,从而使数据和视图变化的相互影响最小。本章将对 InterView 框架的组成部分进行详细介绍。

15.1 模型/视图结构与 MVC 设计模式

MVC 设计模式是起源于 Smalltalk 的一种与用户界面相关的设计模式。MVC 包括三个元素:模型 (Model) 表示数据,视图 (View) 是用户界面,控制 (Controller) 定义了用户在界面上的操作。通过使用 MVC 模式,有效地分离了数据和用户界面,使得设计更为灵活,更能适应变化。

与 MVC 模式不同的是,Qt 的 InterView 框架中把视图和控制部件结合在一起,这称为模型/视图结构。这种结构还是将数据和显示分离了,但框架更为简洁。为了灵活地处理用户输入,InterView 引入了代理 (delegate) 的概念。通过使用代理,能够自定义数据条目 (item) 的显示和编辑方式。图 15-1 展示了 Qt 的模型/视图结构。

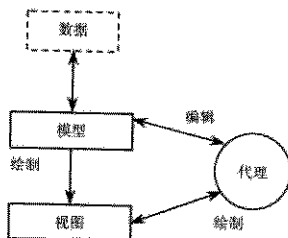


图 15-1 InterView 结构

Qt 的模型/视图结构分为三部分:模型、视图和代理。模型和数据源通信,并为其他部件提供接口。视图从模型中获得模型索引 (Model Index),模型索引用来引用数据条目。在视图中,代理负责绘制数据条目,当编辑条目时,代理和模型直接进行通信。模型/视图/代理之间通过信号和槽进行通信,其关系如下:

- 数据发生改变时,模型发出信号通知视图;
- 用户对界面进行了操作,视图发出信号;
- 代理发出信号告知模型和视图编辑器目前的状态。

15.1.1 模型

所有的模型都基于 `QAbstractItemModel` 类,该类是抽象基类。图 15-2 显示了 Qt 中模型类的继承

关系。

由图 15-2 可以看出, `QAbstractListModel` 和 `QAbstractTableModel` 是列表和表格模型的抽象基类, 如果要实现列表或表格模型, 则应该从这两个类继承。`QAbstractProxyModel` 类是代理模型的抽象类, 后面会讲到。其他的类则是可以直接使用的模型类, 如 `QDirModel` 提供了文件和目录的存储模型, `QStringListModel` 完成 `QStringList` 的存储等。 `QSqlQueryModel` 及其子类是有关数据库的, 在数据库的部分将会讲解。在具体实现中, 相关的数据可以存在模型中, 也可以存在独立的类或外部文件、数据库中。

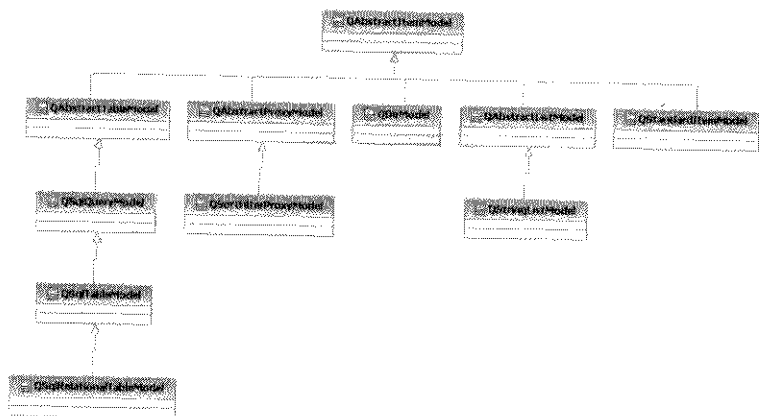


图 15-2 棒型举端承关系

15.1.2 视图

InterView 框架中所有的视图类都从抽象基类 `QAbstractItemView` 继承，图 15-3 显示了视图类之间的继承关系。

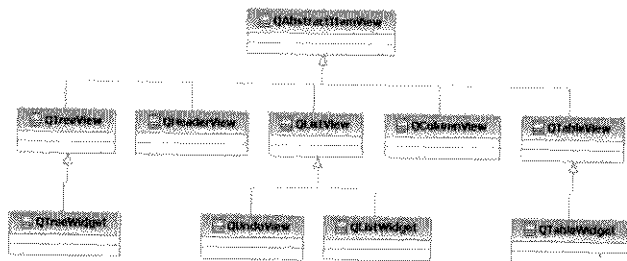


图 15-3 视图类继承关系

Qt 实现了 5 种基本的视图类，并提供了三种不用和模型关联的 Widget 类，以及 Undo Framework 中使用的 QUndoView 类。其中的 QTreeWidget, QListWidget 和 QTableWidget 实际上已经包含了数据，是模型视图集成在一起的类。



15.1.3 代理

InterView 框架中的代理基于 `QAbstractItemDelegate` 抽象类，并提供了一个通用的实现类 `QItemDelegate`，如图 15-4 所示。

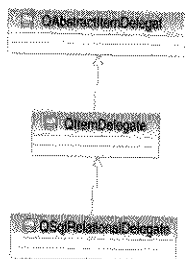


图 15-4 代理类继承关系

图中的 `QSqlRelationalDelegate` 是数据库中的关系代理，在数据库的部分会有相应介绍。

如果需要改变条目的编辑行为和显示，则应该定义自己的代理类。只有需要对视图做比较大的变动时，才需要自定义视图类。

15.2 使用已有的模型视图类

15.2.1 使用已有的模型和视图类

InterView 框架提供了一些常见的模型类和视图类，例如 `QStandardItemModel`, `QDirModel`, `QStringListModel` 和 `QColumnView`, `QHeaderView`, `QListView`, `QTableView`, `QTreeView`。一般情况下可以直接使用这些类来完成一些功能。

下面将实现一个简单的目录浏览程序来体验 `QDirModel` 和 `QTreeView` 的使用，在程序里显示了磁盘上的所有目录和文件，并能够通过编辑框输入目录或文件直接定位相应的项目。首先给出视图窗口部件类定义。

```

#ifndef DIRWIDGET_H_
#define DIRWIDGET_H_

#include <QtGui>

class DirWidget : public QWidget
{
    Q_OBJECT

public:
    DirWidget();
    virtual ~DirWidget();

private slots:

```



```

void pathChanged();

private:
    QModelIndex index;
    QVBoxLayout *layout;
    QDirModel *model;
    QTreeView *tree;
    QLineEdit *dirEdit;
    QCompleter *completer;
};

```

```
#endif /* DIRWIDGET_H_ */
```

在 `DirWidget` 类中，有一个 `QTreeView` 类和一个 `QLineEdit` 类。其中 `QCompleter` 是自动完成类，它能够完成对输入的自动提示。

在构造函数中，对模型和视图进行了初始化和关联。

```

DirWidget::DirWidget()
{
    model = new QDirModel;
    tree = new QTreeView;
    tree->setModel(model);
    index = model->index(QDir::currentPath());
    tree->expand(index);
    tree->scrollTo(index);
    tree->header()->setResizeMode(QHeaderView::ResizeToContents);

    completer = new QCompleter(this);
    completer->setModel(model);
    dirEdit = new QLineEdit;
    dirEdit->setText(QDir::currentPath());
    dirEdit->setCompleter(completer);
    connect(dirEdit, SIGNAL(editingFinished()), this,
            SLOT(pathChanged()));

    layout = new QVBoxLayout;
    layout->addWidget(tree);
    layout->addWidget(dirEdit);
    setLayout(layout);

    resize(640, 480);
    setWindowTitle(QObject::tr("目录浏览"));
}

```

上面的代码中获取了当前的目录，并使用 `QTreeView` 的 `expand()` 函数展开当前目录，`scrollTo()` 函数则使视图滚动到当前目录，使当前目录可见，然后 `setResizeMode(QHeaderView::ResizeToContents)` 使得树视图的显示空间足够宽，能够看到完整的目录名。

对于编辑框 `dirEdit` 对象，设置了自动完成功能。自动完成类 `QCompleter` 也是基于模型的，这里



使用了 QDirModel，这样编辑框将能自动判断用户键入的目录和文件。

当用户输入完路径后，将更新 QTreeView。

```
void DirWidget::pathChanged()
{
    index = model->index(dirEdit->text());
    tree->expand(index);
    tree->scrollTo(index);
}
```

程序运行效果如图 15-5 所示。

15.2.2 QListWidget、QTreeWidget 和 QTableWidgetItem

相对于使用现有的模型和视图，Qt 还提供了更为便捷的类来处理常见的一些数据模型。这些类是 QListWidget、QTreeWidget 和 QTableWidgetItem。它们将模型和视图合一，便于处理一些常规的数据类型。使用这些类虽然简便，但也失去了模型/视图结构的灵活性，所以要根据具体情况来选用。

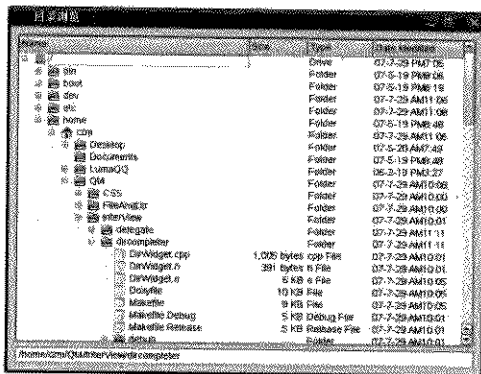


图 15-5 QDirModel 示例

下面以 QTableWidgetItem 类为例来学习如何使用这些现成的类，在例子中使用了 Qt 的 Undo 框架来完成编辑和格式设置的撤消/重做。Undo 框架是用来实现撤消/重做过程的通用框架，使用它简化了实现撤消/重做的代码量。

首先定义一个从 QTableWidgetItem 继承的 UndoWidget 类，该类能实现修改单元格的前景色、背景色、字体，并能撤消和重做这些操作。

```
#ifndef UNDO_WIDGET_H_
#define UNDO_WIDGET_H_

#include <QTableWidgetItem>
#include <QUndoStack>

class UndoWidget : public QTableWidgetItem
{
    ...
}
```

```

Q_OBJECT

public:
    QUndoStack *undoStack;

    UndoWidget(QWidget * parent = 0);
    virtual ~UndoWidget();

public slots:
    void undo();
    void redo();

protected slots:
    void changeFrgColor();
    void changeBkgColor();
    void changeFont();
    void recordText(QTableWidgetItem *current,
        QTableWidgetItem* /*previous*/);
    void itemEdited(QTableWidgetItem* item);

private:
    QAction *actFrgColor;
    QAction *actBkgColor;
    QAction *actFont;
    QString currentText;
    bool ignore;

    void populateTable();
    void createActions();
};
#endif /* UNDO_WIDGET_H */

```

类中的 QUndoStack 是 Undo 框架中提供存放撤消/重做的堆栈。QUndoStack 存放 QUndoCommand 对象。QUndoCommand 命令可以通过 push 压入堆栈，undo()和 redo()函数执行撤消和重做。

在类的实现中，首先初始化表格，然后进行信号和槽的连接。

```

#include <QtGui>
#include "UndoWidget.h"
#include "commands.h"

UndoWidget::UndoWidget(QWidget * parent)
: QTableWidget(parent)
{
    ignore = false;
    setRowCount(5);
    setColumnCount(5);
    createActions();
    populateTable();
    undoStack = new QUndoStack;
}

```



```

connect(this,
        SIGNAL(currentItemChanged(QTableWidgetItem*, QTableWidgetItem*)),
        this, SLOT(recordText(QTableWidgetItem*, QTableWidgetItem*)));
connect(this, SIGNAL(itemChanged(QTableWidgetItem*)),
        this, SLOT(itemEdited(QTableWidgetItem*)));
}

```

ignore 变量用来排除不必要的编辑命令压栈操作。因为产生文字编辑命令压栈操作是监听 itemChanged() 信号, 而在 undo、redo 操作以及设置单元格格式时也会产生这个信号, 但这时是不需要生成新的文字编辑命令的, 所以用 ignore 变量来过滤这些不必要的操作。

通过在 QTableWidget 上弹出右键菜单来显示格式修改命令, 其功能函数 createActions() 如下:

```

void UndoWidget::createActions()
{
    actFrgColor = new QAction(tr("文字颜色"), this);
    connect(actFrgColor, SIGNAL(triggered()),
            this, SLOT(changeFrgColor()));

    actBkgColor = new QAction(tr("背景颜色"), this);
    connect(actBkgColor, SIGNAL(triggered()),
            this, SLOT(changeBkgColor()));

    actFont = new QAction(tr("字体"), this);
    connect(actFont, SIGNAL(triggered()), this, SLOT(changeFont()));

    addAction(actFrgColor);
    addAction(actBkgColor);
    addAction(actFont);
    setContextMenuPolicy(Qt::ActionsContextMenu);
}

```

函数中将上下文菜单的策略设为 Qt::ActionsContextMenu, 说明上下文菜单项从 QWidget::actions() 获得。

选择了菜单后, 需要响应菜单项的格式修改命令, 其中改变文字颜色的槽函数如下:

```

void UndoWidget::changeFrgColor()
{
    QTableWidgetItem *item = currentItem();
    QColor oldColor = item ? item->textColor() : palette().base().color();
    QColor textcolor = QColorDialog::getColor(oldColor, this);
    if (!textcolor.isValid())
        return;

    QList<QTableWidgetItem*> selected = selectedItems();
    if (selected.count() == 0)
        return;

    foreach(QTableWidgetItem *item, selected)
        if (item) {

```

```

        ignore = true;
        item->setTextColor(textColor); // deprecated, use setForeground()
    }
    undoStack->push(new TextColorCommand(this, oldColor, textColor));
}

```

这里取得了 `QTableWidgetItem` 改变前的颜色和改变后的颜色，并创建了 `TextColorCommand` 命令将其压入 `Undo` 堆栈。代码中的 `selectedItems()` 返回所有选中的单元格对象，这里有一个问题，就是当选中多个单元格时，没有分别判断每个单元格原来的文字颜色，因此在撤销/重做时不一定正确，有兴趣的读者可以修改一下代码，完善这个功能。`setTextColor()` 是已经过时的函数，所以在 `Qt Assistant` 中是找不到的，`Qt 4.3` 推荐使用 `setForeground()`。通过 `setForegroud()` 不仅可以设置颜色，还可以设置纹理，这里为了简单起见还是用了 `setTextColor()`。

改变背景的函数代码如下：

```

void UndoWidget::changeBkgColor()
{
    QTableWidgetItem *item = currentItem();
    // QTableWidgetItem 首次返回的背景色是黑色，导致撤销时背景是黑色
    QColor oldColor = item ? item->background().color() :
        palette().base().color();
    QColor bkgColor = QColorDialog::getColor(oldColor, this);
    if (!bkgColor.isValid())
        return;

    QList<QTableWidgetItem*> selected = selectedItems();
    if (selected.count() == 0)
        return;

    foreach(QTableWidgetItem *item, selected)
        if (item) {
            ignore = true;
            item->setBackgroundColor(bkgColor);
        }
    undoStack->push(new BkgColorCommand(this, oldColor, bkgColor));
}

```

对 `QTableWidgetItem` 类如果最开始没有设置背景色，则首次取得的背景色是黑色（这应该是一个 Bug），所以撤销后单元格背景将变为黑色。



扩展阅读

从 `QTableWidgetItem` 的构造函数中，可以看出 `QTableWidget` 没有对 `Qt::BackgroundRole` 或 `Qt::BackgroundRole` 初始化。`QTableWidgetItem` 的 `data()` 函数如下：

```

QVariant QTableWidgetItem::data(int role) const
{
    role = (role == Qt::EditRole ? Qt::DisplayRole : role);
    for (int i = 0; i < values.count(); ++i)

```



```
        if (values.at(i).role == role)
            return values.at(i).value;
        return QVariant();
    }
```

从函数中可以看出, 对于没有初始化的数据直接返回了空值, 这就是对 `QTableWidgetItem` 调用 `background()` 函数返回黑色的原因。

改变字体的函数如下:

```
void UndoWidget::changeFont()
{
    QList<QTableWidgetItem*> selected = selectedItems();
    if (selected.count() == 0)
        return;
    bool ok = false;
    QFont oldFont = currentItem()->font();
    QFont font = QFontDialog::getFont(&ok, oldFont, this);
    if (!ok)
        return;
    foreach(QTableWidgetItem *item, selected)
        if (item) {
            ignore = true;
            item->setFont(font);
        }

    undoStack->push(new FontCommand(this, oldFont, font));
}
```

由于 `QTableWidget` 和 `QTableWidgetItem` 没有提供获取编辑之前文字的方法, 所以用 `recordText()` 随时记录当前文字。

```
void UndoWidget::recordText(QTableWidgetItem *current, QTableWidgetItem* /*previous*/)
{
    if(current)
        currentText = current->text();
    else
        currentText.clear();
}
```

用户改变单元格后, 产生文字编辑命令 `EditCommand`, 这里需要过滤掉一些不必要压栈的响应。

```
void UndoWidget::itemEdited(QTableWidgetItem *item)
{
    if(ignore) // 避免 Undo、Redo 操作等再次产生压栈命令
    {
        ignore = false;
        return;
    }
    undoStack->push(new EditCommand(item, currentText, item->text()));
}
```

undo 和 redo 操作如下:

```
void UndoWidget::undo()
{
    ignore = true;
    undoStack->undo();
}

void UndoWidget::redo()
{
    ignore = true;
    undoStack->redo();
}
```

这里一共定义了 4 个 **QUndoCommand** 类, 分别是文字颜色设置、背景色设置、字体设置和文字编辑函数, 它们定义如下:

```
#ifndef COMMANDS_H
#define COMMANDS_H

#include <QUndoCommand>
#include <QTableWidget>
#include <QColor>
#include <QList>

class UndoWidget;

class TextColorCommand : public QUndoCommand
{
public:
    TextColorCommand(UndoWidget *undoWidget, QColor beforeColor,
                     QColor afterColor, QUndoCommand *parent = 0);

    void undo();
    void redo();

private:
    QList<QTableWidgetItem*> affectedItems;
    QColor before;
    QColor after;
};

class BkgColorCommand : public QUndoCommand
{
public:
    BkgColorCommand(UndoWidget *undoWidget, QColor beforeColor,
                     QColor afterColor, QUndoCommand *parent = 0);

    void undo();
}
```



```
void redo();

private:
    QList<QTableWidgetItem*> affectedItems;
    QColor before;
    QColor after;
};

class FontCommand : public QUndoCommand
{
public:
    FontCommand(UndoWidget *undoWidget, QFont beforeFont, QFont afterFont,
                QUndoCommand *parent = 0);

    void undo();
    void redo();

private:
    QList<QTableWidgetItem*> affectedItems;
    QFont before;
    QFont after;
};

class EditCommand : public QUndoCommand
{
public:
    EditCommand(QTableWidgetItem *item, QString beforeText,
                QString afterText, QUndoCommand *parent = 0);

    void undo();
    void redo();

private:
    QTableWidgetItem* affectedItem;
    QString before;
    QString after;
};

#endif
```

每个撤消命令都定义了撤消/重做操作必需的数据，它们的实现如下：

```
#include <QtGui>
#include "commands.h"
#include "undowidget.h"

TextColorCommand::TextColorCommand(UndoWidget *undoWidget,
    QColor beforeColor, QColor afterColor, QUndoCommand *parent)
    : QUndoCommand(parent)
```



```

{
    affectedItems = undoWidget->selectedItems();
    before = beforeColor;
    after = afterColor;
    setText(QObject::tr("设置文字颜色"));
}

void TextColorCommand::undo()
{
    foreach(QTableWidgetItem *item, affectedItems)
        if (item)
            item->setTextColor(before);    // deprecated, use setForeground()
}

void TextColorCommand::redo()
{
    foreach(QTableWidgetItem *item, affectedItems)
        if (item)
            item->setTextColor(after);    // deprecated, use setForeground()
}

BkgColorCommand::BkgColorCommand(UndoWidget *undoWidget,
    QColor beforeColor, QColor afterColor, QUndoCommand *parent)
    : QUndoCommand(parent)
{
    affectedItems = undoWidget->selectedItems();
    before = beforeColor;
    after = afterColor;
    setText(QObject::tr("设置背景颜色"));
}

void BkgColorCommand::undo()
{
    foreach(QTableWidgetItem *item, affectedItems)
        if (item)
            item->setBackgroundColor(before);
}

void BkgColorCommand::redo()
{
    foreach(QTableWidgetItem *item, affectedItems)
        if (item)
            item->setBackgroundColor(after);
}

FontCommand::FontCommand(UndoWidget *undoWidget, QFont beforeFont,
    QFont afterFont, QUndoCommand *parent)
    : QUndoCommand(parent)

```

```

    {
        affectedItems = undoWidget->selectedItems();
        before = beforeFont;
        after = afterFont;
        setText(QObject::tr("设置字体:") + afterFont.family());
    }

void FontCommand::undo()
{
    foreach(QTableWidgetItem *item, affectedItems)
        if (item)
            item->setFont(before);
}

void FontCommand::redo()
{
    foreach(QTableWidgetItem *item, affectedItems)
        if (item)
            item->setFont(after);
}

EditCommand::EditCommand(QTableWidgetItem *item, QString beforeText,
    QString afterText, QUndoCommand *parent)
    : QUndoCommand(parent)
{
    affectedItem = item;
    before = beforeText;
    after = afterText;
    setText(QObject::tr("输入文字:") + after);
}

void EditCommand::undo()
{
    if (affectedItem)
        affectedItem->setText(before);
}

void EditCommand::redo()
{
    if (affectedItem)
        affectedItem->setText(after);
}

```

撤消命令中 `setText()` 函数用于设置 `undo` 命令在 `QUndoView` 中显示的文字。
最后给出主窗口的定义:

```

#ifndef MAINWINDOW_H_
#define MAINWINDOW_H_

```

```

#include <QMainWindow>
#include "undowidget.h"

class MainWindow : public QMainWindow
{
public:
    MainWindow(QWidget * parent = 0, Qt::WindowFlags flags = 0);
    virtual ~MainWindow();

private:
    UndoWidget *undoWidget;
    QAction *actUndo;
    QAction *actRedo;
    QToolBar *editToolBar;
    QUndoView *undoView;

    void createActions();
    void createToolbars();
};

#endif /*MAINWINDOW_H*/

```

主窗口在构造函数中创建了一个 QUndoView 窗口，该窗口能够显示撤消/重做命令列表，其实现代码如下：

```

#include <QtGui>
#include "mainwindow.h"

MainWindow::MainWindow(QWidget * parent, Qt::WindowFlags flags)
: QMainWindow(parent, flags)
{
    undoWidget = new UndoWidget(this);
    resize(640,480);
    createActions();
    createToolbars();

    undoView = new QUndoView(undoWidget->undoStack);
    undoView->setWindowTitle(tr("命令列表"));
    undoView->setAttribute(Qt::WA_QuitOnClose, false);
    undoView->show();

    // setCentralWidget(undoWidget);
    setWindowTitle(tr("可撤销编辑的 TableWidget"));
}

void MainWindow::createActions()
{

```

```

    actUndo = new QAction(QIcon(":/images/undo.png"), tr("撤销"), this);
    actUndo->setEnabled(false);
    actRedo = new QAction(QIcon(":/images/redo.png"), tr("重做"), this);
    actRedo->setEnabled(false);
    connect(actUndo, SIGNAL(triggered()), undoWidget, SLOT(undo()));
    connect(actRedo, SIGNAL(triggered()), undoWidget, SLOT(redo()));
    connect(undoWidget->undoStack, SIGNAL(canRedoChanged(bool)),
            actRedo, SLOT(setEnabled(bool)));
    connect(undoWidget->undoStack, SIGNAL(canUndoChanged(bool)),
            actUndo, SLOT(setEnabled(bool)));
}

void MainWindow::createToolbars()
{
    editToolBar = addToolBar(tr("编辑"));
    editToolBar->addAction(actUndo);
    editToolBar->addAction(actRedo);
}

```

main.cpp 的实现如下, 因为使用了标准的颜色和字体对话框, 这些对话框默认是显示为英文界面, 所以需要加载 Qt 的简体中文翻译文件, 使其显示为中文。

```

#include <QtGui>
#include "mainwindow.h"

int main(int argc, char** argv)
{
    Q_INIT_RESOURCE(undo);
    QApplication app(argc, argv);

    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
    QTranslator translator;
    QStringList environment = QProcess::systemEnvironment();
    QString str;
    foreach(str, environment) {
        if(str.startsWith("QTDIR="))
            break;
    }
    if(str.startsWith("QTDIR="))
    {
        QString qtdir = str.mid(6);
        translator.load("qt_" + QLocale::system().name(),
                       qtdir.append("/translations/"));
        qApp->installTranslator(&translator);
    }
    else {
        qDebug("You must set QTDIR environment variable.");
        return 1;
    }
}

```

```

MainWindow mainWin;
mainWin.show();
return app.exec();
}

```

在主程序中，通过查找 QTDIR 变量找到 Qt 的安装目录，然后在安装目录下找到相应的语言文件（中文的是 qt_zh_CN.qm）。程序的运行界面如图 15-6 所示。

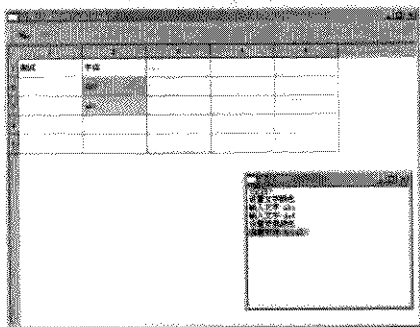


图 15-6 具备撤消/重做功能的 QTableWidget

15.3 模型 (Models)

15.3.1 模型索引

模型为视图和代理提供了标准的接口。Qt 中所有模型都是基于 QAbstractItemModel 派生的。当数据改变时，模型发出信号通知视图。为了保证数据的存取和表示分离，InterView 引入了模型索引 (Model Index) 的概念。每个信息条目 (Item) 通过模型索引来获取，视图和代理使用索引来存取数据。

通过模型索引来存取数据条目，必须有三个属性：行号、列号和父条目的模型索引。对于不同的模型索引组织可能不一样，下面是三种常见模型索引的组织方式。

表模型索引在同一层次，只需要指定行和列就能确定一个元素的位置，如图 15-7 所示。

列表模型类似于表模型，只不过列表模型只有一列。树模型顶层的树枝节点的行号是连续的，每一树枝下的同层节点行号重新编排，因此在存取树索引时，不但要指定行、列，还要指定父索引，其组织方式如图 15-8 所示。

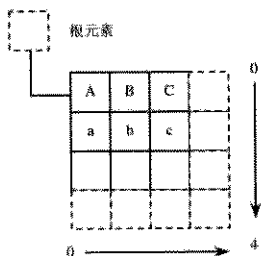


图 15-7 表模型索引组织方式

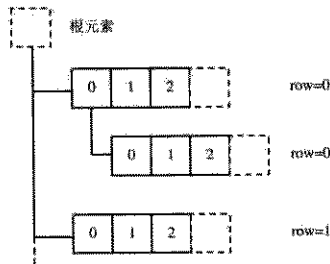


图 15-8 树模型索引组织方式



模型索引只是提供了临时索引信息,因为模型可能对内部的结构进行重新组织,所以模型索引可能在下次使用时失效。如果索引要长期引用,可以使用 Qt 的 `QPersistentModelIndex` 类保存模型索引。

15.3.2 模型角色

模型中的条目能够有不同的角色,这样可以在不同的情况下提供不同的数据。例如 `Qt::DisplayRole` 用来存取视图中显示的文字,角色由枚举类型 `Qt::ItemDataRole` 定义,表 15-1 列出了主要的角色及其描述。

表 15-1 Item 的角色

常 量	描 述
<code>Qt::DisplayRole</code>	显示文字
<code>Qt::DecorationRole</code>	绘制装饰数据 (通常是图标)
<code>Qt::EditRole</code>	在编辑器中编辑的数据
<code>Qt::ToolTipRole</code>	工具提示
<code>Qt::StatusTipRole</code>	状态栏提示
<code>Qt::WhatsThisRole</code>	What's This 文字
<code>Qt::SizeHintRole</code>	尺寸提示
<code>Qt::FontRole</code>	默认代理绘制使用的字体
<code>Qt::TextAlignmentRole</code>	默认代理的对齐方式
<code>Qt::BackgroundRole</code>	默认代理的背景画刷
<code>Qt::ForegroundRole</code>	默认代理的前景画刷
<code>Qt::CheckStateRole</code>	默认代理的检查框状态
<code>Qt::UserRole</code>	用户自定义的数据的起始位置

例如,要给单元格设置工具提示信息,可以使用语句:

```
model->setData(index, tr("tooltip"), Qt::ToolTipRole);
```

15.3.3 自定义模型

如果要实现自己的模型,可以从通用的 `QAbstractItemModel` 类继承,也可以从类 `QAbstractListModel` 和 `QAbstractTableModel` 继承实现列表模型或表格模型。实现自定义模型必须要实现基类的纯虚函数和其他一些特定的函数。

在数据库中为了避免冗余,通常需要将一些重复的文字字段使用代码保存,然后通过外键关联操作来查找其真实的含义。这里将实现一个将数值代码转换为文字的模型来学习如何实现自定义模型,为了不偏离主题,这里没有直接使用数据库。

在模型中存储了不同军种的各种武器,模型的定义如下:

```
#ifndef WEAPONMODEL_H
#define WEAPONMODEL_H

#include <QAbstractTableModel>
#include <QModelIndex>
#include <QVariant>
```

```

class WeaponModel : public QAbstractTableModel
{
public:
    WeaponModel(QObject *parent = 0);
    virtual int rowCount(const QModelIndex& parent = QModelIndex()) const;
    virtual int columnCount(const QModelIndex& parent = QModelIndex()) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation, int role)
        const;

private:
    QVector<short> army;
    QVector<short> weapon;
    QStringList type;
    QMap<short, QString> armyMap;
    QMap<short, QString> weaponMap;
    QStringList header;

    void populateModel();
};
#endif

```

这里从 `QAbstractTableModel` 继承, 实现了基本的纯虚函数 `rowCount()`, `columnCount()`, `data()`, 以及返回表头数据的 `headerData()` 函数。

首先初始化表格, 并填充数据。

```

#include <QtGui>
#include "weaponmodel.h"

WeaponModel::WeaponModel(QObject *parent)
: QAbstractTableModel(parent)
{
    armyMap[1] = tr("空军");
    armyMap[2] = tr("海军");
    armyMap[3] = tr("陆军");
    armyMap[4] = tr("海军陆战队");

    weaponMap[1] = tr("轰炸机");
    weaponMap[2] = tr("战斗机");
    weaponMap[3] = tr("航空母舰");
    weaponMap[4] = tr("驱逐舰");
    weaponMap[5] = tr("直升机");
    weaponMap[6] = tr("坦克");
    weaponMap[7] = tr("两栖攻击舰");
    weaponMap[8] = tr("两栖战车");

    populateModel();
}

```



这里使用了 QMap 数据结构来存储“数值-文字”的映射。populateModel() 完成表格数据的初始填充。

```
void WeaponModel::populateModel()
{
    header << tr("军种") << tr("种类") << tr("武器");
    army << 1 << 2 << 3 << 4 << 2 << 4 << 3 << 1;
    weapon << 1 << 3 << 5 << 7 << 4 << 8 << 6 << 2;
    type << tr("B-2") << tr("尼米兹级") << tr("阿帕奇") << tr("黄蜂级")
        << tr("阿利伯克级") << tr("AAAV") << tr("M1A1") << tr("F-22");
}
```

模型的列固定为 3，所以直接返回“3”。

```
int WeaponModel::columnCount(const QModelIndex& parent) const
{
    return 3;
}
```

rowCount() 返回模型的行数。

```
int WeaponModel::rowCount(const QModelIndex& parent) const
{
    return army.size();
}
```

data() 函数返回指定索引的数据，在这里将数值映射成文字返回。

```
QVariant WeaponModel::data(const QModelIndex &index, int role) const
{
    if(!index.isValid())
        return QVariant();
    qDebug() << index;
    if(role == Qt::DisplayRole)
    {
        switch(index.column())
        {
            case 0:
                return armyMap[army[index.row()]];
                break;
            case 1:
                return weaponMap[weapon[index.row()]];
                break;
            case 2:
                return type[index.row()];
            default:
                return QVariant();
        }
    }
    return QVariant();
}
```


headerData()函数返回固定的表头数据，这里只设置水平表头的标题：

```
QVariant WeaponModel::headerData(int section, Qt::Orientation orientation, int role) const
{
    if(role == Qt::DisplayRole && orientation == Qt::Horizontal)
        return header[section];
    return QAbstractTableModel::headerData(section, orientation, role);
}
```

最后是启动文件 `main.cpp`，在这里将模型和视图关联。

```
#include <QtGui>
#include "weaponmodel.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

    WeaponModel model;

    QTableView view;
    view.setModel(&model);
    view.setWindowTitle(QObject::tr("自定义模型"));
    view.resize(640, 480);
    view.show();
    return app.exec();
}
```

映射的效果如图 15-9 所示。

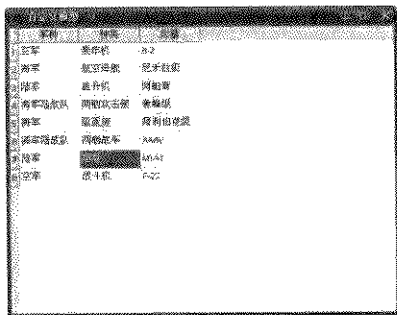


图 15-9 代码转换自定义模型

15.3.4 代理模型

如果想对模型的数据进行处理，例如进行排序或过滤，可以使用代理模型（Proxy Model）。代理模型提供了模型和视图之间的数据处理代理，使得源模型和视图之间的数据能够进行一些特殊处理。

Qt 提供了 `QSortFilterProxyModel` 代理模型完成排序和过滤操作。如果想自己定义代理模型，可以从 `QSortFilterProxyModel` 继承完成一些排序过滤操作，也可以直接从 `QAbstractItemModel` 模型继承完成代理模型。

下面将通过实现一个过滤代理模型的程序来学习代理模型的使用。在例子中学生的考试成绩可以按不同分数段进行过滤显示。首先用 Qt 设计器设计如图 15-10 所示的界面。

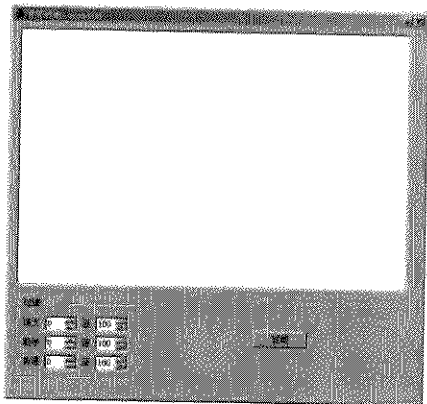


图 15-10 过滤模型用户界面

如图 15-10 所示，最上面的是 `QTableView` 窗口部件，对象名是 `tableView`。需要将该控件的 `sortingEnabled` 属性设为“true”，使得表格能够排序。其中的每个 `QSpinBox` 的取值范围都是 0~100，左边一列初始值为 0，右边一列初始值为 100。

接下来设计过滤类。

```
#ifndef MYSORTFILTERPROXYMODEL_H
#define MYSORTFILTERPROXYMODEL_H

#include <QSortFilterProxyModel>

class MySortFilterProxyModel : public QSortFilterProxyModel
{
    Q_OBJECT

public:
    MySortFilterProxyModel(QObject *parent = 0);

    void setMinChinese(const int chinese);
    void setMaxChinese(const int chinese);

    void setMinMath(const int math);
    void setMaxMath(const int math);

    void setMinEnglish(const int english);
```

```

void setMaxEnglish(const int english);

protected:
    bool filterAcceptsRow(int sourceRow, const QModelIndex &sourceParent) const;

private:
    int minChinese;
    int maxChinese;
    int minMath;
    int maxMath;
    int minEnglish;
    int maxEnglish;
};

#endif

```

在过滤类中提供了设置过滤语文、数学、英语成绩范围的函数 `filterAcceptsRow()`，该函数对于需要过滤的值返回“false”。

接着再来看过滤类的实现文件，首先是设置范围的6个函数。

```

#include <QtGui>
#include "mysortfilterproxymodel.h"

MySortFilterProxyModel::MySortFilterProxyModel(QObject *parent)
    : QSortFilterProxyModel(parent)
{
}

void MySortFilterProxyModel::setMinChinese(const int chinese)
{
    minChinese = chinese;
    invalidateFilter();
}

void MySortFilterProxyModel::setMaxChinese(const int chinese)
{
    maxChinese = chinese;
    invalidateFilter();
}

void MySortFilterProxyModel::setMinMath(const int math)
{
    minMath = math;
    invalidateFilter();
}

void MySortFilterProxyModel::setMaxMath(const int math)
{
    maxMath = math;
}

```

```

        invalidateFilter();
    }

void MySortFilterProxyModel::setMinEnglish(const int english)
{
    minEnglish = english;
    invalidateFilter();
}

void MySortFilterProxyModel::setMaxEnglish(const int english)
{
    maxEnglish = english;
    invalidateFilter();
}

```

每个函数都调用了 `invalidateFilter()` 函数，该函数告诉模型过滤条件已经改变，需要更新代理模型并调用 `filterAcceptsRow()` 函数。

```

bool MySortFilterProxyModel::filterAcceptsRow(int sourceRow,
        const QModelIndex &sourceParent) const
{
    QModelIndex indexChinese = sourceModel()->index(sourceRow, 1, sourceParent);
    QModelIndex indexMath = sourceModel()->index(sourceRow, 2, sourceParent);
    QModelIndex indexEnglish = sourceModel()->index(sourceRow, 3, sourceParent);

    int chinese = sourceModel()->data(indexChinese).toInt();
    if(chinese < minChinese || chinese > maxChinese)
        return false;
    int math = sourceModel()->data(indexMath).toInt();
    if(math < minMath || math > maxMath)
        return false;
    int english = sourceModel()->data(indexEnglish).toInt();
    if(english < minEnglish || english > maxEnglish)
        return false;

    return true;
}

```

`filterAcceptsRow()` 针对模型的每一行进行调用。通过判断每一列的值是否在过滤条件内，如果是则过滤掉（返回“false”），否则返回“true”。

接下来定义界面显示的类。

```

#ifndef SCORE_H_
#define SCORE_H_

#include <QWidget>
#include <QStandardItemModel>
#include "ui_score.h"

class MySortFilterProxyModel;

```

```

class Score : public QWidget, private Ui_score
{
    Q_OBJECT

public:
    Score(QWidget * parent = 0, Qt::WindowFlags f = 0);
    virtual ~Score();

protected slots:
    void setFilter();

protected:
    void populateTable();

    MySortFilterProxyModel *proxyModel;
    QStandardItemModel *model;
};

```

```
#endif /*SCORE_H*/
```

该类的实现文件完成数据的初始化和过滤条件的设置。

```

#include <QtGui>
#include "Score.h"
#include "mysortfilterproxymodel.h"

Score::Score(QWidget * parent, Qt::WindowFlags f)
: QWidget(parent, f)
{
    setupUi(this);
    populateTable();
    connect(btnFilter, SIGNAL(clicked(bool)), this, SLOT(setFilter()));

    proxyModel = new MySortFilterProxyModel(this);
    proxyModel->setDynamicSortFilter(true);

    proxyModel->setSourceModel(model);
    tableView->setAlternatingRowColors(true);
    tableView->setModel(proxyModel);
    setFilter();
}

```

在构造函数中使用代理模型的 `setSourceModel()` 函数设置代理模型的源模型，视图则使用该代理模型。

当用户单击过滤按钮时，`setFilter()` 函数设置模型的过滤条件，并调用视图的 `update()` 函数更新视图，其实现如下所示。

```

void Score::populateTable()
{

```



```
model = new QStandardItemModel(3, 4, this);

model->setHeaderData(0, Qt::Horizontal, tr("姓名"));
model->setHeaderData(1, Qt::Horizontal, tr("语文"));
model->setHeaderData(2, Qt::Horizontal, tr("数学"));
model->setHeaderData(3, Qt::Horizontal, tr("英语"));

model->setData(model->index(0, 0), tr("李逵"));
model->setData(model->index(0, 1), 65);
model->setData(model->index(0, 2), 68);
model->setData(model->index(0, 3), 70);

model->setData(model->index(1, 0), tr("武松"));
model->setData(model->index(1, 1), 70);
model->setData(model->index(1, 2), 75);
model->setData(model->index(1, 3), 83);

model->setData(model->index(2, 0), tr("宋江"));
model->setData(model->index(2, 1), 88);
model->setData(model->index(2, 2), 90);
model->setData(model->index(2, 3), 93);
}

void Score::setFilter()
{
    proxyModel->setMinChinese(spinChineseFrom->value());
    proxyModel->setMaxChinese(spinChineseTo->value());
    proxyModel->setMinMath(spinMathFrom->value());
    proxyModel->setMaxMath(spinMathTo->value());
    proxyModel->setMinEnglish(spinEnglishFrom->value());
    proxyModel->setMaxEnglish(spinEnglishTo->value());
    tableView->update(); // 如果不使用 update(), 表格的列标题将显示混乱
}
```

当用户点击“过滤”按钮时，setFilter()函数设置模型的过滤条件，并调用视图的 update()函数更新视图。

15.4 视图 (Views)

15.4.1 自定义视图

通常情况下，如果要使用定制的视图，可以通过实现条目代理 (Item Delegate) 来完成。只有需要对视图进行彻底改变时，才需要从 QAbstractItemView 继承来实现自己的视图。这里不详细讲解，Qt 中有一个例子 chart，展示了如何实现自定义视图，需要自定义视图的读者可以参考。

15.4.2 数据-窗口部件映射

用过 Access 数据库的用户应该知道，Access 数据库能够提供一条记录在一个表单上显示编辑，并

能够进行前后移动的功能，Qt 的 `QDataWidgetMapper` 类就提供了类似于 Access 这样的功能。通过使用 `QDataWidgetMapper` 类，可以将模型一行中每一列的数据映射到窗口部件，让特定的窗口部件显示和编辑模型的数据。

下面的例子展示了如何将模型数据映射到窗口部件。先给出整个表单的定义，这个表单中包含了模型中映射的各种窗口部件，包括 `QComboBox`, `QLineEdit`, `QDoubleSpinBox`, `QLabel`。

```
#ifndef MAPWIDGET_H
#define MAPWIDGET_H

#include <QtGui>

class MapWidget : public QWidget
{
    Q_OBJECT

public:
    MapWidget(QWidget *parent = 0);

private slots:
    void updateUI(int row);
    inline void toNext()
    {
        mapper->toNext();
    };

    inline void toPrevious()
    {
        mapper->toPrevious();
    };

private:
    void populateModel();

    QLabel *companyLabel;
    QLabel *typeLabel;
    QLabel *displacementLabel;
    QLabel *nationLabel;

    QComboBox *companyCombo;
    QLineEdit *typeEdit;
    QDoubleSpinBox *displacementSpin;
    QLabel *flagLabel;

    QPushButton *btnNext;
    QPushButton *btnPrev;
    QPushButton *btnSubmit;
    QPushButton *btnRevert;
```



```
QStringList companies;

QStandardItemModel *model;
QDataWidgetMapper *mapper;

QShortcut *leftShortCut;    // 上一条快捷键
QShortcut *rightShortCut;   // 下一条快捷键
);
#endif
```

表单中提供了查看下一条、上一条记录的按钮，还有提交和放弃修改的按钮，用来控制数据导航和修改。

构造函数对界面进行初始化。

```
#include <QtGui>
#include "mapwidget.h"
#include "mapdelegate.h"

MapWidget::MapWidget(QWidget *parent)
    : QWidget(parent)
{
    populateModel();

    companyLabel = new QLabel(tr("公司"));
    typeLabel = new QLabel(tr("车型"));
    displacementLabel = new QLabel(tr("排量"));
    nationLabel = new QLabel(tr("原产国"));

    companyCombo = new QComboBox;
    companyCombo->addItem(companies);
    typeEdit = new QLineEdit;
    displacementSpin = new QDoubleSpinBox;
    displacementSpin->setSingleStep(0.1);
    flagLabel = new QLabel;

    btnNext = new QPushButton(tr("下一条"));
    btnPrev = new QPushButton(tr("前一条"));
    btnSubmit = new QPushButton(tr("提交"));
    btnRevert = new QPushButton(tr("放弃"));

    mapper = new QDataWidgetMapper(this);
    mapper->setSubmitPolicy(QDataWidgetMapper::ManualSubmit);
    mapper->setModel(model);
    mapper->addMapping(companyCombo, 0);
    mapper->addMapping(typeEdit, 1);
    mapper->addMapping(displacementSpin, 2);
    QByteArray property("pixmap");
```



```

mapper->addMapping(flagLabel, 3, property);
MapDelegate *delegate = new MapDelegate(this);
mapper->setItemDelegate(delegate);

connect(btnPrev, SIGNAL(clicked()), mapper, SLOT(toPrevious()));
connect(btnNext, SIGNAL(clicked()), mapper, SLOT(toNext()));
connect(btnSubmit, SIGNAL(clicked()), mapper, SLOT(submit()));
connect(btnRevert, SIGNAL(clicked()), mapper, SLOT(revert()));
connect(mapper, SIGNAL(currentIndexChanged(int)),
        this, SLOT(updateUI(int)));

QGridLayout *layout = new QGridLayout();
layout->addWidget(companyLabel, 0, 0);
layout->addWidget(companyCombo, 0, 1, 1, 2);
layout->addWidget(typeLabel, 1, 0);
layout->addWidget(typeEdit, 1, 1, 1, 2);
layout->addWidget(displacementLabel, 2, 0);
layout->addWidget(displacementSpin, 2, 1, 1, 2);
layout->addWidget(nationLabel, 3, 0);
layout->addWidget(flagLabel, 3, 1, 1, 2);

layout->addWidget(btnPrev, 4, 0);
layout->addWidget(btnNext, 4, 1);
layout->addWidget(btnSubmit, 4, 2);
layout->addWidget(btnRevert, 4, 3);
setLayout(layout);

leftShortCut = new QShortcut(QKeySequence::MoveToNextLine,
                             this, SLOT(toNext()));
rightShortCut = new QShortcut(QKeySequence::MoveToPreviousLine,
                              this, SLOT(toPrevious()));

setWindowTitle(tr("数据映射"));
mapper->toFirst();
}

```

QDataWidgetMapper 类对数据提交有两种策略：QDataWidgetMapper::AutoSubmit 是默认策略，该策略自动保存修改的数据。QDataWidgetMapper::ManualSubmit 是手动提交，在数据修改后需要手动确认修改。这里使用了手动提交，读者可以将其改为自动提交进行比较。

在上面的映射过程中，因为国家要用国旗的图标显示，所以在映射到 QLabel 时要特别指明 QLabel 的 pixmap 属性。

其中公司的字段使用了 QComboBox 对象，这需要使用一个代理才能正确地映射数据到 QComboBox，后面将要讲到这个代理类。

建立用户界面后，需要初始化数据。

```

void MapWidget::populateModel()
{

```



```

model = new QStandardItemModel(7, 4, this);

companies << tr("奇瑞") << tr("标致雪铁龙") << tr("福特") << tr("本田")
    << tr("大众") << tr("现代") << tr("菲亚特");

QStringList types;
types << tr("旗云") << tr("标致 307") << tr("福克斯") << tr("思域")
    << tr("宝来") << tr("伊兰特") << tr("西耶那");

QList<qreal> displacements;
displacements << 1.3 << 2.0 << 2.0 << 1.6 << 1.8 << 1.6 << 1.5;

QList<QPixmap> flags;
flags << QPixmap(":/images/china.png") << QPixmap(":/images/france.png")
    << QPixmap(":/images/usa.png") << QPixmap(":/images/japan.png")
    << QPixmap(":/images/germany.png") << QPixmap(":/images/korea.png")
    << QPixmap(":/images/italy.png");

QStandardItem *item;
for (int row = 0; row < 7; ++row) {
    item = new QStandardItem(companies[row]);
    model->setItem(row, 0, item);
    item = new QStandardItem(types[row]);
    model->setItem(row, 1, item);
    model->setData(model->index(row, 2, QModelIndex()),
        displacements[row]);
    model->setData(model->index(row, 3, QModelIndex()), flags[row]);
}
}

```

根据数据在开头或末尾，将前一条或后一条的按钮禁止。

```

void MapWidget::updateUI(int row)
{
    btnPrev->setEnabled(row > 0);
    btnNext->setEnabled(row < model->rowCount() - 1);
}

```

前面使用 `QComboBox` 来映射公司名，但公司名如果不使用代理，是不能正确地映射到 `QComboBox` 的。所以这里实现了一个条目的代理类。

```

#ifndef MAPDELEGATE_H
#define MAPDELEGATE_H

#include <QItemDelegate>
#include <QModelIndex>
#include <QObject>

class MapDelegate : public QItemDelegate

```

```

{
    Q_OBJECT

public:
    MapDelegate(QObject *parent = 0);

    void setEditorData(QWidget *editor, const QModelIndex &index) const;
    void setModelData(QWidget *editor, QAbstractItemModel *model,
        const QModelIndex &index) const;

};

#endif

```

在代理类中，主要是实现 `setEditorData()` 将模型的数据映射到 `QComboBox`，实现 `setModeldata()` 函数将 `QComboBox` 选择的项映射到模型中。

```

#include <QtGui>
#include "mapdelegate.h"

MapDelegate::MapDelegate(QObject *parent)
    : QItemDelegate(parent)
{
}

void MapDelegate::setEditorData(QWidget *editor,
    const QModelIndex& index) const
{
    if(index.column() == 0) // company
    {
        QComboBox *comboEditor = qobject_cast<QComboBox *>(editor);
        if (comboEditor) {
            int i = comboEditor->findText(
                index.model()->data(index, Qt::EditRole).toString());
            comboEditor->setCurrentIndex(i);
        }
    }
    else
        return QItemDelegate::setEditorData(editor, index);
}

void MapDelegate::setModelData(QWidget *editor, QAbstractItemModel *model,
    const QModelIndex& index) const
{
    if(index.column() == 0)
    {
        QComboBox *comboBox = qobject_cast<QComboBox *>(editor);
        if (comboBox)
        {

```

```

        model->setData(index, comboBox->currentText());
    }
    else
        return QItemDelegate::setModelData(editor, model, index);
}

```

最后完成主程序 main.cpp，可以看到如图 15-11 所示的效果。



图 15-11 模型映射窗口部件示例

15.5 代理 (Delegates)

和 MVC 模式不同的是，Qt 的模型/视图没有单独用来管理用户交互的部件，通常用户交互的工作由视图来完成。如果需要灵活的用户交互方式，可以使用代理 (Delegates) 来完成。代理负责管理条目的用户输入和条目的绘制。

Qt 使用抽象类 `QAbstractItemDelegate` 作为代理类的基类。用户实现代理类时，至少要实现两个纯虚函数，其中 `paint()` 函数完成绘制，`sizeHint()` 函数返回条目的尺寸。大部分情况下，可以直接从 Qt 已经实现的通用 `QItemDelegate` 类继承，以减少代码量。

15.5.1 使用已有的代理

Qt 实现了一个基本的代理类 `QItemDelegate`，在 `QListView`、`QTableView`、`QTreeView` 中都使用了这个代理类。`QItemDelegate` 提供了显示和编辑模型数据条目的基本功能。对于不同的数据类型，`QItemDelegate` 提供了默认的编辑方式。也可以通过 `QItemEditorFactory` 类来注册特定数据类型的编辑器。

15.5.2 自定义代理

如果要在编辑条目时使用特定的控件，可以自定义代理。下面将建立一个示例，分别对不同的列使用 `QDateTimeEdit`、`QSpinBox` 和 `QComboBox` 控件进行编辑。代理类定义如下：

```

#ifndef DELEGATE_H
#define DELEGATE_H

#include <QItemDelegate>
#include <QModelIndex>
#include <QObject>
#include <QSpinBox>

class EditorDelegate : public QItemDelegate

```

```

{
    Q_OBJECT

public:
    EditorDelegate(QObject *parent = 0);

    QWidget *createEditor(QWidget *parent,
        const QStyleOptionViewItem &option,
        const QModelIndex &index) const;

    void setEditorData(QWidget *editor, const QModelIndex &index) const;
    void setModelData(QWidget *editor, QAbstractItemModel *model,
        const QModelIndex &index) const;

    void updateEditorGeometry(QWidget *editor,
        const QStyleOptionViewItem &option, const QModelIndex &index) const;
};
#endif

```

通常都从 `QItemDelegate` 类继承，而不从抽象类 `QAbstractDelegateItem` 类继承，以减少工作量。在这里要实现几个函数：`createEditor()`函数创建特定的编辑控件；`setEditorData()`函数给编辑控件设定初始数据；`setModelData()`函数在编辑完成后将数据写入 `Model`；`updateEditorGeometry()`函数管理编辑控件的位置。接下来实现该类。

```

#include <QtGui>
#include "delegate.h"

EditorDelegate::EditorDelegate(QObject *parent)
    : QItemDelegate(parent)
{
}

QWidget *EditorDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem& option,
    const QModelIndex& index) const
{
    if(index.column() == 1)    // birthday
    {
        QDateTimeEdit *editor = new QDateTimeEdit(parent);
        editor->setDisplayFormat("yyyy/M/dd");
        editor->setCalendarPopup(true);
        return editor;
    }
    else if(index.column() == 2)    // height
    {
        QSpinBox *editor = new QSpinBox(parent);
        editor->setMinimum(140);
        editor->setMaximum(200);
    }
}

```



```
        return editor;
    }
    else if(index.column() == 3)    // province
    {
        QComboBox *editor = new QComboBox(parent);
        editor->addItem(tr("北京"));
        editor->addItem(tr("天津"));
        editor->addItem(tr("上海"));
        editor->addItem(tr("湖北"));
        editor->addItem(tr("山东"));
        editor->addItem(tr("江西"));
        editor->addItem(tr("河北"));
        editor->addItem(tr("江苏"));
        editor->addItem(tr("浙江"));
        return editor;
    }
    else
        return QItemDelegate::createEditor(parent, option, index);
}
```

在 `createEditor()` 函数中, 根据模型中的列不同使用了不同的控件, 并对窗口部件进行了初始化。例如初始化了 `QDateTimeEdit` 窗口部件的日期显示格式, `QSpinBox` 窗口部件的上下限, 以及 `QComboBox` 窗口部件中的条目。

创建窗口部件后, 需要将模型中的数据通过 `setEditor()` 函数传给编辑窗口部件, 如下代码完成这个过程。

```
void EditorDelegate::setEditorData(QWidget *editor,
                                   const QModelIndex& index) const
{
    if(index.column() == 1)    // birthday
    {
        QDateTimeEdit *dateEditor =
            qobject_cast<QDateTimeEdit *>(editor);
        if (dateEditor) {
            dateEditor->setDate(
                QDate::fromString(index.model()->data(
                    index, Qt::EditRole).toString(),
                    "yyyy/M/d"));
        }
    }
    else if(index.column() == 2)    // height
    {
        QSpinBox *spinEditor = qobject_cast<QSpinBox *>(editor);
        if(spinEditor)
            spinEditor->setValue(index.model()->data(index,
                Qt::EditRole).toInt());
    }
    else if(index.column() == 3)    // province
```

```

{
    QComboBox *comboBox = qobject_cast<QComboBox *>(editor);
    if (comboBox)
    {
        int i = comboBox->findText(
            index.model()->data(index, Qt::EditRole).toString());
        comboBox->setCurrentIndex(i);
    }
}
else
    return QItemDelegate::setEditorData(editor, index);
}

```

上面的函数使用 `qobject_cast` 对编辑窗口部件进行转换，转换成功后将编辑窗口部件的数据设置为模型中相应的数据。

编辑结束后，应该使用 `setModelData()` 将编辑控件的数据写入到模型中。

```

void EditorDelegate::setModelData(QWidget *editor,
    QAbstractItemModel *model,
    const QModelIndex& index) const
{
    if (index.column() == 1) // birthday
    {
        QDateTimeEdit *dateEditor =
            qobject_cast<QDateTimeEdit *>(editor);
        if (dateEditor) {
            model->setData(index,
                dateEditor->date().toString("yyyy/M/d"));
        }
    }
    else if (index.column() == 2) // height
    {
        QSpinBox *spinEditor = qobject_cast<QSpinBox *>(editor);
        if (spinEditor)
            model->setData(index,
                QString("%1").arg(spinEditor->value()));
    }
    else if (index.column() == 3) // province
    {
        QComboBox *comboBox = qobject_cast<QComboBox *>(editor);
        if (comboBox)
        {
            model->setData(index, comboBox->currentText());
        }
    }
    else
        return QItemDelegate::setModelData(editor, model, index);
}

```

这也是对象的转换和编辑控件值的转换过程。最后是更新编辑窗口部件位置的函数。

```
void EditorDelegate::updateEditorGeometry(QWidget *editor,
    const QStyleOptionViewItem &option, const QModelIndex & /* index */) const
{
    editor->setGeometry(option.rect);
}
```

option.rect 属性中保存了条目的位置，这里将编辑控件设置在刚好占住条目的位置。最后完成主程序并初始化数据。

```
#include <QtGui>
#include "delegate.h"

void populateData(QTableWidget& table)
{
    table.setItem(0, 0, new QTableWidgetItem(QObject::tr("林黛玉")));
    table.setItem(1, 0, new QTableWidgetItem(QObject::tr("史湘云")));
    table.setItem(2, 0, new QTableWidgetItem(QObject::tr("薛宝钗")));
    table.setItem(3, 0, new QTableWidgetItem(QObject::tr("贾元春")));

    table.setItem(0, 1, new QTableWidgetItem(QObject::tr("1980/2/3")));
    table.setItem(1, 1, new QTableWidgetItem(QObject::tr("1981/10/24")));
    table.setItem(2, 1, new QTableWidgetItem(QObject::tr("1978/5/15")));
    table.setItem(3, 1, new QTableWidgetItem(QObject::tr("1979/8/29")));

    table.setItem(0, 2, new QTableWidgetItem(QObject::tr("165")));
    table.setItem(1, 2, new QTableWidgetItem(QObject::tr("168")));
    table.setItem(2, 2, new QTableWidgetItem(QObject::tr("162")));
    table.setItem(3, 2, new QTableWidgetItem(QObject::tr("163")));

    table.setItem(0, 3, new QTableWidgetItem(QObject::tr("江苏")));
    table.setItem(1, 3, new QTableWidgetItem(QObject::tr("浙江")));
    table.setItem(2, 3, new QTableWidgetItem(QObject::tr("江西")));
    table.setItem(3, 3, new QTableWidgetItem(QObject::tr("湖北")));
}

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

    QTableWidget table(3,4);
    QStringList labels;
    labels << QObject::tr("姓名") << QObject::tr("出生日期")
        << QObject::tr("身高") << QObject::tr("籍贯");
    table.setHorizontalHeaderLabels(labels);

    EditorDelegate delegate;
```



```

table.setItemDelegate(&delegate);

populateData(table);

table.setWindowTitle(QObject::tr("代理示例"));
table.resize(800, 600);
table.show();
return app.exec();
}

```

主程序中使用 `setItemDelegate()` 函数设置了 `QTableWidget` 的代理。从图 15-12 可以看到使用 `QDateEdit` 编辑窗口部件的效果。

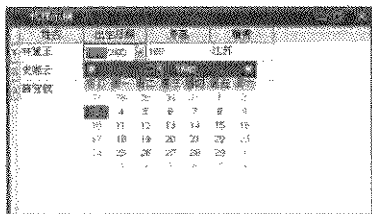


图 15-12 视图代理

15.6 拖放与选中

15.6.1 拖放操作

`ImmerView` 框架支持内部和外部的拖放操作，但要在模型的实现中实现相应的函数。如果需要简单步骤就能支持拖放，可以使用 `QListWidget`、`QTableWidget` 和 `QTreeWidget`，如下的代码打开了 `QTableWidget` 的拖放操作。

```

QTableWidget *tableWidget = new QTableWidget(this);
tableWidget->setSelectionMode(QAbstractItemView::SingleSelection);
tableWidget->setDragEnabled(true);
tableWidget->setAcceptDrops(true);
tableWidget->setDropIndicatorShown(true);

```

如果在模型视图结构的应用中打开拖放功能，则要稍微复杂一些。除了要像上面一样设置视图的属性外，还要实现模型的一些函数。下面实现一个基于 `QAbstractTableModel` 的模型来说明如何实现拖放操作。

模型定义如下：

```

#ifndef TABLEMODEL_H
#define TABLEMODEL_H

#include <QAbstractTableModel>
#include <QStringList>

```



```

class TableModel : public QAbstractTableModel
{
    Q_OBJECT

public:
    TableModel(QObject* parent = 0);
    int rowCount(const QModelIndex & parent = QModelIndex()) const;
    int columnCount(const QModelIndex & parent = QModelIndex()) const;
    QVariant data(const QModelIndex & index,
        int role = Qt::DisplayRole) const;
    bool setData(const QModelIndex & index, const QVariant & value,
        int role = Qt::EditRole);
    Qt::DropActions supportedDropActions() const;
    Qt::ItemFlags flags(const QModelIndex & index) const;
    QStringList mimeTypes() const;
    QMimeData* mimeData(const QModelIndexList & indexes) const;
    bool dropMimeData(const QMimeData *data, Qt::DropAction action,
        int row, int column, const QModelIndex & parent);

private:
    void populateTable();
    QStringList nameList;
    QStringList emperorList;
    QStringList dynastyList;
    QStringList yearList;
};
#endif

```

在类 `TableModel` 中, `rowCount()`, `columnCount()` 和 `data()` 是必须要重载的纯虚函数, 其他重载函数如 `supportedDropActions()`, `flags()` 等都是为了支持拖放操作而实现的。

在 `supportedDropActions()` 函数中, 返回模型支持的拖放操作。

```

Qt::DropActions TableModel::supportedDropActions() const
{
    return Qt::CopyAction | Qt::MoveAction;
}

```

实际上, 当模型的拖放模式设为 `QAbstractItemView::InternalMove` 时, 拖放操作总是 `Qt::MoveAction`。

同时还要重载 `QAbstractItemModel::flags()` 函数, 以允许拖放。

```

Qt::ItemFlags TableModel::flags(const QModelIndex & index) const
{
    Qt::ItemFlags defaultFlags = QAbstractTableModel::flags(index);

    if (index.isValid())
        return Qt::ItemIsDragEnabled | Qt::ItemIsDropEnabled | defaultFlags;
    else
        return Qt::ItemIsDropEnabled | defaultFlags;
}

```

当拖放条目时, 使用 `QMimeData` 对象存储拖放数据, 这里只用纯文本。

```
QStringList TableModel::mimeTypes() const
{
    QStringList types;
    types << "text/plain";
    return types;
}
```

开始拖动时, `mimeData()` 函数用来设置数据。

```
QMimeData *TableModel::mimeData(const QModelIndexList &indexes) const
{
    QMimeData *mimeData = new QMimeData();

    QModelIndex idx = indexes[0];
    if (idx.isValid()) {
        QString text = data(idx, Qt::DisplayRole).toString();
        mimeData->setText(text);
    }

    return mimeData;
}
```

因为只允许单选, 所以在函数中只取第一个元素。

数据放下时, `dropMimeData()` 数据负责设置数据。

```
bool TableModel::dropMimeData(const QMimeData *data,
    Qt::DropAction action, int row, int column, const QModelIndex &parent)
{
    if (action == Qt::IgnoreAction)
        return true;

    if (!data->hasFormat("text/plain"))
        return false;

    if (parent.isValid())
        setData(parent, data->text());
    else
        return false;

    return true;
}
```

由于要设置数据, 因此 `setData()` 函数也必须要实现。

```
bool TableModel::setData(const QModelIndex & index, const QVariant & value,
    int role)
{
    int row = index.row();
    if (row > rowCount())
```

```

    {
        return false;
    }
    switch(index.column())
    {
    case 0:
        nameList[row] = value.toString();
        break;
    case 1:
        emperorList[row] = value.toString();
        break;
    case 2:
        dynastyList[row] = value.toString();
        break;
    case 3:
        yearList[row] = value.toString();
        break;
    }
    return true;
}

```

最后在主程序中还要设置 QTableView 的拖放属性。

```

#include <QtGui>
#include "tablemodel.h"

int main(int argv, char **args)
{
    QApplication app(argv, args);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

    TableModel tableModel;
    QTableView view;
    view.setModel(&tableModel);
    view.setSelectionMode(QAbstractItemView::SingleSelection);
    view.setDragDropMode(QAbstractItemView::InternalMove);
    view.setDragEnabled(true);
    view.setAcceptDrops(true);
    view.setDropIndicatorShown(true);
    view.resize(640, 480);
    view.show();

    return app.exec();
}

```

上面实现的程序对拖放到表格外面的数据没有处理，读者可以自行添加此功能，这要实现 QAbstractItemModel::insertRows() 和 QAbstractItemModel::insertColumns() 函数。

15.6.2 选中模式

模型/视图中选中条目使用 QItemSelectionModel 类来管理，它只和模型相关，和视图无关，这样

可以使不同视图中的选择同步。

选择由选择范围组成 (Selection Ranges), 非连续的选择由多个选择范围组成。一个选择范围由模型的起始元素和结束元素的索引组成。选择范围使用 `QItemSelectionRange` 类来管理, 多个选择范围可以使用基于 `QList` 的 `QItemSelection` 类管理。

一个视图的选择模型可以通过 `setSelectionModel()` 函数来设置, 函数 `selectionModel()` 可以返回模型的选择范围。当选择发生改变时, `QItemSelectionModel` 发出 `selectionChanged` 信号。

对于 `QTableWidget` 等控件, 选择方法则有些不同。例如 `QTableWidget` 使用 `QTableWidgetSelectionRange` 类来管理选择, 或者使用 `QTableWidgetItem` 的 `setSelected()` 函数对单个单元格设置选中状态。

用户的选择方式可以使用 `QAbstractItemView::SelectionMode` 属性来控制, 有如表 15-2 所示的几种选择模式。

表 15-2 视图的选择模式

选择模式	描 述
<code>QAbstractItemView::SingleSelection</code>	单个选择模式, 当用户选择条目时, 之前选中的将被取消
<code>QAbstractItemView::ContiguousSelection</code>	连续选择模式, 用户可以使用 Shift 加鼠标连续选择或拖动鼠标连续选择, 但不能进行不连续的选择
<code>QAbstractItemView::ExtendedSelection</code>	扩展选择模式, 即可以使用 Shift 和鼠标进行连续选择(或拖动鼠标), 也可以使用 Ctrl 和鼠标进行不连续的选择, 这是默认的模式
<code>QAbstractItemView::MultiSelection</code>	多选, 用户选择条目时不影响其他已选中的条目
<code>QAbstractItemView::NoSelection</code>	用户不能选择条目

可以通过 `selectionBehavior` 属性控制用户选择的行为, 有如表 15-3 所示的几种行为。

表 15-3 `QAbstractItemView` 的选择行为

选择行为	描 述
<code>QAbstractItemView::SelectItems</code>	选择单个的条目
<code>QAbstractItemView::SelectRows</code>	只能选择行
<code>QAbstractItemView::SelectColumns</code>	只能选择列

15.7 小 结

模型/视图结构是 Qt 4 中类似于 MVC 的一种框架, 在 Qt 中是以模型、视图、代理的方式实现的。最简单的应用是直接利用 `QTableWidget`、`QListWidget` 和 `QTreeWidget`, 但这样也失去了模型/视图的灵活性。要充分发挥模型视图的灵活性, 应该使用模型/视图/代理的结构。用户还可以通过自定义模型、代理和视图来实现特定的功能。

第 16 章 高级绘图

为了满足高级绘图的需要,在 Qt 中支持 OpenGL 接口和矢量图形格式 SVG。使用 OpenGL 可以完成 3D 绘图,SVG 模块则可以绘制 XML 格式的矢量图形 SVG。Qt 还支持直接操作显存的 QGLFramebufferObject 对象。

16.1 3D 绘图——使用 OpenGL

OpenGL 是最开始由 SGI 公司推出的 3D 图形编程接口,OpenGL 具有绘制三维图形的各种函数,但不包含窗口系统或处理用户输入的函数。为了在 Qt 中使用 OpenGL,Qt 提供了 OpenGL 模块。在 Qt 中使用 OpenGL 的途径是使用 QGLWidget 类,并使用 OpenGL API 进行绘制。要在 Qt 程序中使用 OpenGL API,需要包含头文件:

```
#include <QtOpenGL>
```

同时在 qmake 的工程文件中加入:

```
QT += opengl
```

由于 OpenGL 内容非常多,关于 OpenGL 的讲解需要独立的一本书,所以本章只介绍在 Qt 中使用 OpenGL 的入门知识。如需要对 OpenGL 有更深入的了解,可以参考相关的 OpenGL 专业书籍。

16.1.1 创建 OpenGL 窗口

创建 OpenGL 应用程序通常要从 QGLWidget 类继承出自己的类。QGLWidget 类从 QWidget 类继承,提供了在 Qt 应用程序中显示 OpenGL 图形的能力。QGLWidget 类提供了三个虚函数来完成 OpenGL 的绘图任务,initializeGL()完成 OpenGL 环境的初始化,paintGL()绘制 OpenGL 图形,resizeGL()在窗口发生改变时运行。下面将建立一个最简单的 OpenGL 程序。

首先定义 MyGLWidget 类,头文件如下所示。

```
#include <QtGui>
```

```
#include <QtOpenGL>
```

```
class MyGLWidget : public QGLWidget
```

```
{
```

```
Q_OBJECT
```

```
public:
```

```
MyGLWidget(QWidget * parent = 0, const QGLWidget * shareWidget = 0,
```

```
Qt::WindowFlags f = 0);
```

```
~MyGLWidget();
```

```
protected:
    void initializeGL();
    void paintGL();
    void resizeGL(int width, int height);
    void mouseDoubleClickEvent( QMouseEvent * event );
};
```

接下来实现 OpenGL 的初始化函数 initializeGL()如下:

```
void MyGLWidget::initializeGL()
{
    glShadeModel(GL_SMOOTH);
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearDepth(1.0);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
}
```

在该函数中调用的全部都是 OpenGL 函数, 下面进行逐一解释。

glClearColor()函数设置当前清除颜色。glClearColor()的三个参数依次是红色、绿色和蓝色分量。分别表示特定颜色分量的多少, 取值范围为[0, 1]。最后一个参数是 Alpha 值, 取值范围也是[0, 1], 表示透明度。程序中的全零参数表示屏幕使用黑色清除。

接下来的三行都是和深度缓冲区相关的。深度缓冲区负责记录观察点和占据每个像素的物体之间的距离, 它存储了每个像素的深度值, 所以又称为 z 缓冲区。

glClearDepth()告诉 glClear()函数清除深度缓冲区时, 要将深度缓冲区的值设为多少。

glEnable()函数启用 OpenGL 的某项功能, 关闭某项功能使用 glDisable()函数。在程序中, 使用参数 GL_DEPTH_TEST 将启用深度比较并更新深度缓冲区。

glDepthFunc()指定深度比较的方法为 GL_LEQUAL。对于屏幕上的每个像素, 深度缓冲区负责记录观察点和占据这个像素的物体之间的距离。如果指定的测试通过, 源片段的深度值就会替换深度缓冲区中原先存在的值。GL_LEQUAL 指定源片段的 z 值小于或等于深度缓冲区中的对应 z 值, 则通过深度测试。

glHint()启用特定的绘制提示, 该行告诉 OpenGL 进行最好的透视修正。

在这个初始化函数中, 对 OpenGL 进行一些设置。设置了用黑色为清除屏幕色, 并打开深度缓存, 启用平滑着色等。



扩展阅读

OpenGL 的函数都使用了前缀 gl, 并且组成函数名的每个单词的首字母进行大写, 如 glClearColor()。OpenGL 定义的常量使用大写形式, 前缀为 GL, 单词之间使用下划线分隔, 如 GL_DEPTH_TEST。有些函数有不同的形式, 如 glVertex3f()函数的后缀“3f”中的“3”表示函数有三个参数, “f”表示这些参数都是浮点数。

所有的绘图操作都在 paintGL()函数中, 这里将绘制一个三角形和一个五角星, 代码如下:



```

void MyGLWidget::paintGL()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glLoadIdentity();
    glTranslatef( -1.5, 0.0, -6.0 );
    glBegin( GL_TRIANGLES );
        glVertex3f( 0.0, 1.0, 0.0 );
        glVertex3f( -1.0, -1.0, 0.0 );
        glVertex3f( 1.0, -1.0, 0.0 );
    glEnd();
    glTranslatef( 3.0, 0.0, 0.0 );

    glBegin(GL_LINE_LOOP);
        glVertex(1);
        glVertex(4);
        glVertex(2);
        glVertex(0);
        glVertex(3);
    glEnd();
}

```

在函数中，首先调用 `glClear()` 清除屏幕及深度缓冲区。然后调用 `glLoadIdentity()` 装入单位矩阵，执行后，当前点移到了窗口的中心，各坐标轴正方向分别为 *X* 轴从左到右，*Y* 轴从下到上，*Z* 轴从内到外。`glTranslatef()` 函数将坐标原点左移 1.5，向内移 6.0。

绘制三角形的方法是：调用 `glBegin(GL_TRIANGLES)`，并使用 `glVertex3f()` 函数提供三角形的顶点，绘制完毕后调用 `glEnd()`。

五角星则是在构造函数中算出来的五个顶点用直线顺序相连，构造函数如下：

```

MyGLWidget::MyGLWidget(QWidget * parent, const QGLWidget * shareWidget,
                        Qt::WindowFlags f)
{
    setMinimumSize(320,240);
    resize(640,480);
    setWindowTitle(tr("第一个 OpenGL 程序"));
    short angle = 18;
    for(short i=0; i<5; i++) {
        Point[i][0] = cos(angle * PI/180);
        Point[i][1] = sin(angle * PI/180);
        Point[i][2] = 0.0;
        angle += 72;
    }
}

```

窗口大小发生改变时的代码如下：

```

void MyGLWidget::resizeGL(int width, int height)
{
    glViewport( 0, 0, (GLint)width, (GLint)height );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
}

```



```

gluPerspective( 45.0, (GLfloat)width/(GLfloat)height, 0.1, 100.0 );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
}

```

上面代码中, `glMatrixMode(GL_PROJECTION)`指明接下来的函数将影响投影矩阵。投影矩阵负责为场景增加透视。调用 `glLoadIdentity()`后为场景设置透视图。

`glMatrixMode(GL_MODELVIEW)`指明任何新的变换将会影响模型视图矩阵。模型视图矩阵中存放了物体信息。最后重置模型视图矩阵。

`resizeGL()`的作用是重新设置 OpenGL 场景的大小, OpenGL 场景的尺寸将被设置成它显示时所在窗口的大小。

接着实现鼠标双击窗口客户区实现全屏和窗口显示切换。代码如下:

```

void MyGLWidget::mouseDoubleClickEvent( QMouseEvent * event )
{
    if(windowState() & Qt::WindowFullScreen)
        showNormal();
    else
        showFullScreen();
}

```

最后一步是实现 `main.cpp`。

```

#include <QtGui>
#include "mygl.h"

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
    MyGLWidget myglWidget;
    myglWidget.show();
    return app.exec();
}

```

程序的运行效果如图 16-1 所示。

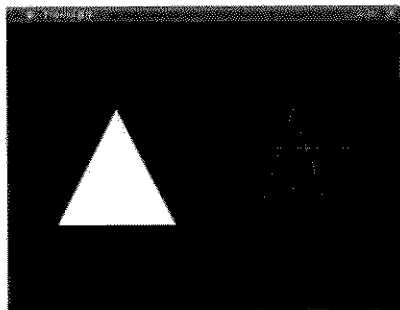


图 16-1 第一个 OpenGL 程序



16.1.2 着色

上节第一个 OpenGL 程序只有黑白两种颜色, 本节将画一个三角形和一个圆, 并对每个顶点着色。在 OpenGL API 中并没有绘制圆形的函数, 只有基本的绘制点、线、三角形、矩形、多边形等方法。要想绘制圆形, 可以用一个边数足够多的多边形来模拟。

着色的方法很简单, 在顶点函数之前调用 `glColor3f()` 函数就可以了。`glColor3f()` 函数在顶点函数之前调用, 它的三个参数依次是红、绿、蓝三种颜色分量, 取值范围是 $[0, 1]$ 。数值越大, 色彩饱和度越高。下面将三角形的每个顶点分别用红、绿、蓝三种颜色着色。圆的每个顶点用饱和度不同的红色着色, 形成渐变效果。

为了绘制圆, 程序用五十边的正多边形来模拟。多边形的顶点需要进行计算, 因此需要定义两个常量, 如下:

```
const short n = 50;
const GLfloat PI = 3.1415926536f;
```

由于使用了 `sin()` 和 `cos()` 函数, 需要包含数学库头文件。

```
#include <math.h>
```

也可以包含标准 C++ 的 `<cmath>` 库, 但要使用名字空间 `std`。绘制函数 `paintGL()` 实现如下:

```
void MyGLWidget::paintGL()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glLoadIdentity();
    glTranslatef( -1.5, 0.0, -5.0 );
    glBegin(GL_TRIANGLES);
        glColor3f( 0.0, 0.0, 1.0 );    // 蓝色
        glVertex3f( 0.0, 1.0, 0.0 );
        glColor3f( 1.0, 0.0, 0.0 );    // 红色
        glVertex3f( -1.0, -1.0, 0.0 );
        glColor3f( 0.0, 1.0, 0.0 );    // 绿色
        glVertex3f( 1.0, -1.0, 0.0 );
    glEnd();
    glTranslatef( 3.0, 0.0, 0.0 );

    glBegin(GL_POLYGON);
        GLfloat part;
        for(short i=0; i<n; ++i) {
            part = 1.0/n * i;
            glColor3f(1.0, part, part);
            glVertex3f(1*cos(2*PI*part), 1*sin(2*PI*part), 0.0);
        }
    glEnd();
}
```

函数中 `glBegin(GL_POLYGON)` 表示开始绘制一个多边形。这里是正五十边多边形, 读者可以将常量 `n` 改为更大或更小的数字, 观察绘制的圆效果。

程序绘制效果如图 16-2 所示。

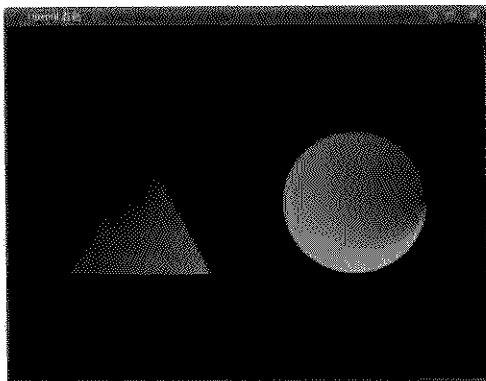


图 16-2 上色的 OpenGL 图形

16.1.3 3D 和旋转

学习了基本的平面图形绘制和着色，接下来使图形成为真正的三维图形，并能够旋转。三维图形的生成实际上就是要生成三维图形的各个面，然后可以绕某个轴进行旋转。类定义修改如下：

```
class MyGLWidget : public QGLWidget
{
    Q_OBJECT

public:
    MyGLWidget(QWidget * parent = 0, const QGLWidget * shareWidget = 0,
                Qt::WindowFlags f = 0);
    ~MyGLWidget();

protected:
    GLfloat rTri;
    GLfloat rQuad;
    GLuint list;

    void initializeGL();
    void paintGL();
    void resizeGL(int width, int height);
    void makeList();
    void timerEvent(QTimerEvent *event);
    void mouseDoubleClickEvent(QMouseEvent * event );
};
```

类中增加了 `rTri` 和 `rQuad` 两个变量用于控制两个三维物体的旋转轴。`list` 变量存储显示列表，`makeList()` 函数生成显示列表。

显示列表表示一组存储在一起的 OpenGL 函数。当调用一个显示列表时，它所存储的函数就会按顺



序执行。大多数 OpenGL 函数可以存储在显示列表中，通过显示列表可以改进性能。下面是生成显示列表的函数：

```
void MyGLWidget::makeList()
{
    list = glGenLists(2);
    glNewList(list, GL_COMPILE);
        glBegin( GL_TRIANGLES );
            glColor3f( 1.0, 0.0, 0.0 );
            glVertex3f( 0.0, 1.0, 0.0 );
            glColor3f( 0.0, 1.0, 0.0 );
            glVertex3f( -1.0, -1.0, 1.0 );
            glColor3f( 0.0, 0.0, 1.0 );
            glVertex3f( 1.0, -1.0, 1.0 );

            glColor3f( 1.0, 0.0, 0.0 );
            glVertex3f( 0.0, 1.0, 0.0 );
            glColor3f( 0.0, 0.0, 1.0 );
            glVertex3f( 1.0, -1.0, 1.0 );
            glColor3f( 0.0, 1.0, 0.0 );
            glVertex3f( 1.0, -1.0, -1.0 );

            glColor3f( 1.0, 0.0, 0.0 );
            glVertex3f( 0.0, 1.0, 0.0 );
            glColor3f( 0.0, 1.0, 0.0 );
            glVertex3f( 1.0, -1.0, -1.0 );
            glColor3f( 0.0, 0.0, 1.0 );
            glVertex3f( -1.0, -1.0, -1.0 );

            glColor3f( 1.0, 0.0, 0.0 );
            glVertex3f( 0.0, 1.0, 0.0 );
            glColor3f( 0.0, 0.0, 1.0 );
            glVertex3f( -1.0, -1.0, -1.0 );
            glColor3f( 0.0, 1.0, 0.0 );
            glVertex3f( -1.0, -1.0, 1.0 );
        glEnd();
    glEndList();

    glNewList(list + 1, GL_COMPILE);
        glBegin( GL_QUADS );
            glColor3f( 0.0, 1.0, 0.0 );
            glVertex3f( 1.0, 1.0, -1.0 );
            glVertex3f( -1.0, 1.0, -1.0 );
            glVertex3f( -1.0, 1.0, 1.0 );
            glVertex3f( 1.0, 1.0, 1.0 );

            glColor3f( 1.0, 0.5, 0.0 );
            glVertex3f( 1.0, -1.0, 1.0 );
```

```

glVertex3f( -1.0, -1.0, 1.0 );
glVertex3f( -1.0, -1.0, -1.0 );
glVertex3f( 1.0, -1.0, -1.0 );

glColor3f( 1.0, 0.0, 0.0 );
glVertex3f( 1.0, 1.0, 1.0 );
glVertex3f( -1.0, 1.0, 1.0 );
glVertex3f( -1.0, -1.0, 1.0 );
glVertex3f( 1.0, -1.0, 1.0 );

glColor3f( 1.0, 1.0, 0.0 );
glVertex3f( 1.0, -1.0, -1.0 );
glVertex3f( -1.0, -1.0, -1.0 );
glVertex3f( -1.0, 1.0, -1.0 );
glVertex3f( 1.0, 1.0, -1.0 );

glColor3f( 0.0, 0.0, 1.0 );
glVertex3f( -1.0, 1.0, 1.0 );
glVertex3f( -1.0, 1.0, -1.0 );
glVertex3f( -1.0, -1.0, -1.0 );
glVertex3f( -1.0, -1.0, 1.0 );

glColor3f( 1.0, 0.0, 1.0 );
glVertex3f( 1.0, 1.0, -1.0 );
glVertex3f( 1.0, 1.0, 1.0 );
glVertex3f( 1.0, -1.0, 1.0 );
glVertex3f( 1.0, -1.0, -1.0 );

glEnd();
glEndList();
}

```

在该函数中，生成了一个显示列表。该显示列表生成了一个金字塔三维物体和一个立方体。OpenGL 中，每个显示列表都有一个整型索引标识，上面代码中的 `glGenLists(2)` 产生两个索引值。然后，`glNewList()` 函数指明开始一个显示列表，`glEndList()` 表示显示列表结束。`glNewList()` 的第二个参数 `GL_COMPILE` 说明仅编译显示列表而不立即执行。

由于使用了显示列表，`paintGL()` 函数变得非常简洁，实现如下：

```

void MyGLWidget::paintGL()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glLoadIdentity();
    glTranslatef( -1.5, 0.0, -6.0 );
    glRotatef( rTri, 0.0, 1.0, 0.0 );
    glCallList(list);

    glLoadIdentity();
    glTranslatef( 1.5, 0.0, -7.0 );
}

```



```

    glRotatef( rQuad, 1.0, 1.0, 1.0 );
    glCallList(list + 1);
}

```

其中 `glRotatef()` 函数的原型是:

```
void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)
```

它的作用是绕从原点到点 (x,y,z) 的向量顺时针旋转 $angle$ 弧度。

最终的程序运行效果如图 16-3 所示。

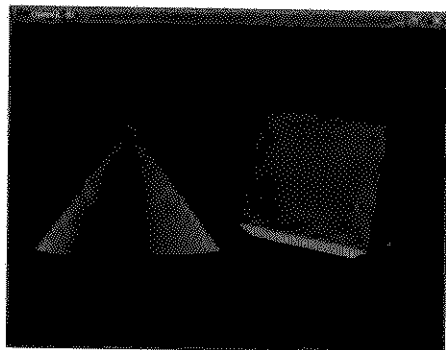


图 16-3 三维旋转物体

16.1.4 纹理贴图

为了让三维图形看上去更真实,可以在三维物体上贴上纹理图像。纹理贴图能够保证当三维物体进行变换和渲染时,纹理也能够表现出正确的行为。纹理通常是二维的,但它也可以是一维的或三维的。这里通过一个简单的例子来学习基本的纹理贴图功能。例子对立方体每一面进行不同的纹理贴图并进行旋转。

首先要在 `initializeGL()` 函数中加入启用二维纹理贴图功能的语句,如下:

```
glEnable( GL_TEXTURE_2D );
```

使用纹理之前,需要创建纹理对象,这里通过函数 `loadGLTextures()` 来实现这个功能。

```

void MyGLWidget::loadGLTextures()
{
    QImage tex, buffer;
    QString fileName(":texture%1.bmp");

    glGenTextures(6, &texture[0]);

    for(short i=0; i<6; i++) {
        buffer.load(fileName.arg(i+1));
        tex = QGLWidget::convertToGLFormat( buffer );
        glBindTexture( GL_TEXTURE_2D, texture[i] );
        glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, tex.width(), tex.height(),

```

```

    0, GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() };
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
}

```

在 loadGLTextures() 函数中装入了一个 QImage 对象作为纹理贴图。由于 OpenGL 并不认识 Qt 的 QImage 图像格式，所以要调用 QGLWidget::convertToGLFormat() 函数将 QImage 对象转换为 OpenGL API 能够识别的图像格式。转换后的 QImage 已经不是 Qt 能识别的格式，但 width(), height() 和 bits() 返回的值仍然是有效的，能够在 OpenGL 应用中使用。

函数 glGenTextures(GLsizei n, GLuint *textureNames) 在数组 textureNames 中返回 n 个当前未使用的值，表示纹理对象的名称。glBindTexture() 函数创建和使用纹理对象。

glTexImage2D() 函数真正地创建纹理。第一个参数 GL_TEXTURE_2D 告诉 OpenGL 此纹理是一个 2D 纹理；第二个参数在只有一种分辨率的纹理贴图的情况下，应该为 0；第三个参数确定了数据的哪些成分（RGBA、深度、亮度）作为图像的纹理单元，这里使用 GL_RGB；tex.width() 是纹理的宽度；tex.height() 是纹理的高度；第 6 个参数是边框的值；GL_RGBA 告诉 OpenGL 纹理图像数据由红、绿、蓝三色数据以及 alpha 通道数据组成；GL_UNSIGNED_BYTE 说明组成图像的数据是无符号字节类型的；最后 tex.bits() 指明了 OpenGL 纹理数据的来源。接下来是语句：

```

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );

```

上面的两行代码告诉 OpenGL 在显示图像时，当图像放大得比原始纹理大（GL_TEXTURE_MAG_FILTER）或缩小得比原始纹理小（GL_TEXTURE_MIN_FILTER）时 OpenGL 采用的滤波方式。通常这两种情况下都采用 GL_LINEAR，这使得纹理从很远处到离屏幕很近时都平滑显示。使用 GL_LINEAR 需要 CPU 和显卡做更多的运算。如果计算机配置较低，可以采用 GL_NEAREST。但过滤的纹理在放大的时候，看起来比较粗糙。当然也可以结合这两种滤波方式。在近处时使用 GL_LINEAR，远处时使用 GL_NEAREST。

接下来在绘图函数 paintGL() 中使用纹理。

```

void MyGLWidget::paintGL()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glLoadIdentity();
    glTranslatef( 0.0, 0.0, -5.0 );
    glRotatef( xRot, 1.0, 0.0, 0.0 );
    glRotatef( yRot, 0.0, 1.0, 0.0 );
    glRotatef( zRot, 0.0, 0.0, 1.0 );

    glBindTexture( GL_TEXTURE_2D, texture[0] );
    glBegin( GL_QUADS );
        // 前面
        glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0, 1.0 );
        glTexCoord2f( 1.0, 0.0 ); glVertex3f( 1.0, -1.0, 1.0 );
        glTexCoord2f( 1.0, 1.0 ); glVertex3f( 1.0, 1.0, 1.0 );
        glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0, 1.0, 1.0 );
    glEnd();
}

```



```
glBindTexture( GL_TEXTURE_2D, texture[1] );
glBegin( GL_QUADS );
    // 后面
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0, 1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( 1.0, 1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( 1.0, -1.0, -1.0 );
glEnd();

glBindTexture( GL_TEXTURE_2D, texture[2] );
glBegin( GL_QUADS );
    // 顶部
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0, 1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, 1.0, 1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( 1.0, 1.0, 1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( 1.0, 1.0, -1.0 );
glEnd();

glBindTexture( GL_TEXTURE_2D, texture[3] );
glBegin( GL_QUADS );
    // 底部
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( 1.0, -1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( 1.0, -1.0, 1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0, 1.0 );
glEnd();

glBindTexture( GL_TEXTURE_2D, texture[4] );
glBegin( GL_QUADS );
    // 右边
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( 1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( 1.0, 1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( 1.0, 1.0, 1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( 1.0, -1.0, 1.0 );
glEnd();

glBindTexture( GL_TEXTURE_2D, texture[5] );
glBegin( GL_QUADS );
    // 左边
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0, 1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0, 1.0, 1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0, 1.0, -1.0 );
glEnd();
}
```

函数中使用 `glTexCoord2f()` 函数设置纹理坐标, `glTexCoord2f()` 的第一个参数是 *X* 坐标, 0.0 是纹

理的左侧, 0.5 是纹理的中点, 1.0 是纹理的右侧。glTexCoord2f() 的第二个参数是 Y 坐标, 0.0 是纹理的底部, 0.5 是纹理的中点, 1.0 是纹理的顶部。

程序运行效果如图 16-4 所示。

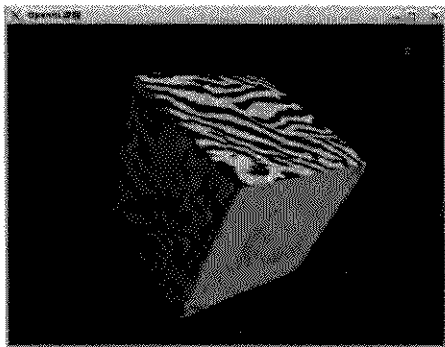


图 16-4 OpenGL 纹理

16.2 SVG

SVG (Scalable Vector Graphics) 是 W3C 开发的基于 XML 的用来描述二维矢量图形和矢量/点阵混合图形的 XML 语言, 是一种全新的矢量图形规范。目前 SVG 1.1 是 W3CSVG 建议的规范, SVG 1.2 正在开发中, 还在草案阶段。SVG 规范定义了 SVG 的特征、语法和显示效果, 包括模块化的 XML 命名空间 (namespace) 和 SVG 文档对象模型 (DOM)。SVG 实现了图形、图像和文字的有机统一。SVG 除了支持 HTML 中常用的标记, 如文本、图像、链接、交互性、CSS 的使用、脚本 (Script) 外, 还提供了大量针对图形、图像、动画的特定标记。

SVG 的 Mobile SVG Profiles (移动 SVG) 包括 SVG Tiny 和 SVG Basic, 面向资源受限的设备, 是第三代移动电话平台的一部分。SVG Basic (Scalable Vector Graphics, Basic Profile) 是 SVG 的一个子集, 其主要目标是为掌上电脑等高端移动设备提供矢量图形显示格式。SVG Tiny (Scalable Vector Graphics, Tiny Profile) 也是 SVG 的一个子集, 其主要目标是为手机等低端移动设备提供矢量图形显示格式。

SVG 主要支持以下几种显示对象:

- 矢量显示对象, 基本矢量显示对象包括矩形、圆、椭圆、多边形、直线、任意曲线等;
- 嵌入式外部图像, 包括 PNG、JPEG、SVG 等;
- 文字对象。

Qt 4 支持 SVG 1.2 Tiny 的静态特性, 支持 SVG 文件的绘制和生成。建立一个 SVG 应用工程需要在 qmake 的 .pro 中加入:

```
Qt += SVG
```

并包含相应的头文件:

```
#include <QtSvg>
```



16.2.1 绘制 SVG 图形

使用 SVG 最简单的方法是构造一个 `QSvgWidget` 对象并使用 `QSvgWidget.load()` 方法装入 SVG 文档,也可以使用 `QSvgRenderer` 类在绘图设备上绘制 SVG 图形。`QSvgRenderer` 类负责在 `QSvgWidget` 上绘制 SVG 图形。当 SVG 文档需要更新时, `QSvgRenderer` 发出 `repaintNeeded` 信号。`QSvgRenderer.render()` 函数负责重绘。

下面实现一个 SVG 文件查看程序,说明 SVG 的使用。定义一个普通的窗口部件如下:

```
class SvgView : public QWidget
{
    Q_OBJECT

public:
    SvgView(const QString &file, QWidget *parent=0);

    virtual QSize sizeHint() const;
protected:
    virtual void paintEvent(QPaintEvent *event);
    virtual void wheelEvent(QWheelEvent *event);

private:
    QSvgRendererer *doc;
};
```

构造函数中对 `QSvgRendererer` 进行初始化。

```
SvgView::SvgView(const QString &file, QWidget *parent)
    : QWidget(parent)
{
    doc = new QSvgRendererer(file, this);
    connect(doc, SIGNAL(repaintNeeded()), this, SLOT(update()));
}
```

信号 `repaintNeeded` 在文档需要更新时发出,通常可用来实现动画效果。

在 `paintEvent()` 函数中, `QSvgRendererer` 对象将 SVG 图形绘制到绘图设备上。

```
void SvgView::paintEvent(QPaintEvent *)
{
    QPainter p(this);
    p.setViewport(0, 0, width(), height());
    doc->render(&p);
}
```

`sizeHint()` 返回 SVG 文档的默认大小。

```
QSize SvgView::sizeHint() const
{
    if (doc)
        return doc->defaultSize();
}
```

```

    return QWidget::sizeHint();
}

```

当鼠标滚轮滚动时，改变 SVG 图形的显示大小。

```

void SvgView::wheelEvent(QWheelEvent *e)
{
    const double diff = 0.1;
    QSize size = doc->defaultSize();
    int width = size.width();
    int height = size.height();
    if (e->delta() > 0) {
        width = int(this->width()+this->width()*diff);
        height = int(this->height()+this->height()*diff);
    } else {
        width = int(this->width()-this->width()*diff);
        height = int(this->height()-this->height()*diff);
    }
    resize(width, height);
}

```

可以看出，由于 Qt 对 SVG 功能的封装，程序员需要做的工作很少。

16.2.2 生成 SVG 文件

在 Qt 4.3 中，引入了 `QSvgGenerator` 类将绘图指令存为 SVG 文件。`QSvgGenerator` 从 `QPaintDevice` 继承，也是一种绘图设备。

使用 `QSvgGenerator` 类十分简单，用 `setFileName()` 指定输出的文件名，然后使用 `QPainter` 的各种绘图函数进行绘制就可以了。下面的程序演示如何生成一个 SVG 文件。

```

#include <QApplication>
#include <QTextCodec>
#include <QtGui>
#include <QtSvg>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

    QSvgGenerator svgGen;
    svgGen.setFileName("test.svg");

    QRectF rectangle(10.0, 20.0, 80.0, 60.0);
    QPainter painter;
    painter.begin(&svgGen);
    QPen pen(Qt::red);
    painter.setPen(pen);
    painter.drawEllipse(rectangle);
    rectangle.moveRight(100);
}

```



```

painter.drawRect(rectangle);

rectangle.moveRight(100);
QRectF source(0.0, 0.0, 70.0, 40.0);
QImage image("Greenstone.bmp");
painter.drawImage(rectangle, image, source);
painter.end();

return 0;
)

```

在上面的例程中，生成了一个 `QSvgGenerator` 对象，然后使用 `setFileName()` 指定了生成的 SVG 文件名，接下来就是使用各种绘图语句在上面绘制，这里绘制了一个椭圆、一个矩形和一幅位图。最终生成的 SVG 文件如下：

```

<?xml version="1.0" standalone="no"?>
<svg width="3.52778cm" height="3.52778cm"
viewBox="0 0 100 100"
xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/ xlink"
version="1.2" baseProfile="tiny">
<title>Qt Svg Document</title>
<desc>Generated with Qt</desc>
<defs>
</defs>
<g fill="none" stroke="black" vector-effect="non-scaling-stroke" stroke-width="1"
fill-rule="evenodd" stroke-linecap="square" stroke-linejoin="bevel">

<g fill="none" stroke="#ff0000" stroke-opacity="1" vector-effect="non-scaling-
stroke" stroke-width="1" stroke-linecap="square" stroke-linejoin="bevel" transform=
"matrix(1,0,0,1,0,0)"
font-family="宋体" font-size="9pt" font-weight="400" font-style="normal"
>
<path fill-rule="evenodd" d="M90,50 C90,66.5685 72.0914,80 50,80 C27.9086,80
10.665685 10,50 C10,33.4315 27.9086,20 50,20 C72.0914,20 90,33.4315 90,50"/>
<path fill-rule="evenodd" d="M20,20 L100,20 L100,80 L20,80 L20,20"/>
<image x="20" y="20" width="80" height="60" xlink:href="data:image/png; base64,
iVBORw0KGgoAAAANS....." />
</g>
</g>
</svg>

```

文件中的图形使用了 base64 编码，由于太长，故省略了 `<image>` 标签中的大量内容。

16.3 小 结

Qt 的 OpenGL 提供了 OpenGL 接口，通过 `QGLWidget` 可以自由地使用 OpenGL 的各种函数实现三维显示。本章只是简单地介绍了 OpenGL 的一些基本用法，更深入的学习可以参考相关的 OpenGL 书籍。Qt 的 SVG 模块支持 SVG 图形的显示和生成，使用也非常简单。

第 17 章 进程与进程间通信

到目前为止，本书的所有程序均是基于单一进程的，而一些较复杂的 Linux 应用程序往往是由多个进程组成的。长期以来，Qt 并未对编写多进程应用程序提供有效支持，可供使用的 QProcess 类也仅仅是把 Linux 底层与进程相关的 API 进行了面向对象的简单封装，进程间通信更是只能依赖原始的 UNIX 进程间通信方法。但从 4.2 版本开始，Qt 引入了一项新的进程间通信技术 D-Bus，并提供了相应的编程模块 QtDBus，从而为在 Qt 下开发多进程应用程序提供了巨大帮助。

17.1 使用 QProcess

Qt 提供了一个 QProcess 类用于启动一个外部程序并与之通信。启动一个新进程的操作十分简单，只需要将待启动的程序名称和启动参数传递给 start() 函数即可。例如：

```
QObject *parent;
...
QString program = "tar";
QStringList arguments;
arguments << "czvf" << "backup.tar.gz"<<"/home";
QProcess *myProcess = new QProcess(parent);
myProcess->start(program, arguments);
```

在调用了 start() 函数后，myProcess 进程立即进入“启动状态”，但 tar 程序尚未被调用，不能读写标准输入输出设备。当进程完成启动后就进入“运行状态”并向外发出 started() 信号。在输入输出方面，QProcess 将一个进程看做一个流类型的 I/O 设备，可以像使用 QTcpSocket 读写流类型的网络连接一样来读写一个进程。可以通过 QIODevice::write() 函数向所启动进程的标准输入写数据，也可以通过 QIODevice::read()、QIODevice::readLine() 和 QIODevice::getChar() 函数从这个进程的标准输出读数据。此外，由于 QProcess 是从 QIODevice 类继承而来的，因此，它也可以作为 QXmlReader 的数据源，或者为 QFtp 产生用于上传的数据。最后，当进程退出时 QProcess 进入起始状态——“非运行状态”，并发出 finished() 信号。

void finished(int exitCode, QProcess::ExitStatus exitStatus) 信号在参数中返回了进程退出的“退出码”和“退出状态”，可以调用 exitCode() 函数和 exitStatus() 函数分别获取最后退出进程的这两个值。其中，Qt 定义的进程“退出状态”只有正常退出和进程崩溃两种，分别对应值 QProcess::NormalExit（值 0）和 QProcess::CrashExit（值 1）。当进程在运行中产生错误时，QProcess 将发出 error() 信号，可以通过调用 error() 函数返回最后一次产生错误的类型，并通过 state() 函数找出此时进程所处状态。其中，Qt 定义的进程错误如表 17-1 所示。



表 17-1 Qt 进程错误表

错误常量	值	描述
<code>QProcess::FailedToStart</code>	0	进程启动失败, 或者是因为被调用的程序缺失, 或者是权限不够
<code>QProcess::Crashed</code>	1	进程成功启动后崩溃
<code>QProcess::Timeout</code>	2	最后一次调用 <code>waitFor...</code> 类型的函数超时。此时 <code>QProcess</code> 状态不变, 并可以再次调用 <code>waitFor...</code> 类型的函数
<code>QProcess::WriteError</code>	3	向进程写入时出错。如进程尚未启动时, 或者输入通道被关闭时
<code>QProcess::ReadError</code>	4	从进程中读取数据时出错。如进程尚未启动时
<code>QProcess::UnknownError</code>	5	未知错误。这也是 <code>error()</code> 函数返回的默认值

对于进程的标准输出这里需要说明一下。进程具有两个预定义的输出通道, 分别是标准输出通道 `stdout`, 通常用于控制台输出, 以及标准错误通道 `stderr`, 通常用于进程打印错误, 它们本质上是两个独立的数据流。可以调用 `setReadChannel()` 函数设置当前的读通道, 当有可读数据时 Qt 将会发出 `readyRead()` 的信号, 如果是标准输出数据时还会发出 `readyReadStandardOutput()` 信号, 如果是标准错误数据也会同时发出 `readyReadStandardError()` 信号。可以分别调用 `readAllStandardOutput()` 函数和 `readAllStandardError()` 函数从标准输出和标准错误通道中读取数据。此外, `QProcess` 还允许将标准输出通道和标准错误通道合并, 两者共用标准输出通道, 方法是在进程启动之前以 `MergedChannels` 参数调用 `setReadChannelMode()` 函数。最后, 进程的输出通道对应 `QProcess` 的读通道, 输入通道对应 `QProcess` 的写通道, 这一点与标准的输入输出术语正好相反, 需要大家引起注意。

下面来看一个实例, 这个例子启动一个 `tar` 进程, 然后打包备份某个路径下的所有内容。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (app.arguments().count() != 2) {
        qDebug() << QObject::tr("需要一个备份路径作为参数, 如: 'process /home'");
        return -1;
    }
    QProcess proc;
    QStringList arguments;
    arguments << "czvf" << "backup.tar.gz";
    arguments << app.arguments().at(1);
    proc.start("tar", arguments);
}
```

这个实例程序必须带有一个备份路径作为参数, 如 “`process /home`”。因此, 首先检查参数个数是否正确, 即 `app.arguments().count()` 是否为 “2”。需要注意, 与 `main()` 函数的参数一样, `arguments()` 函数将程序自身作为第 0 个参数, 并且参数计数也包含了程序自身, 所以这里的 `count()` 值为 “2”。随后将看到取得第一个真正参数的方法是 `app.arguments().at(1)`, 即待备份数据的路径。参数检查通过后, 新建一个 `QProcess` 实例, 并构件 `tar` 程序的参数表 `arguments`, 依次传给它执行选项 “`czvf`”, 存放于当前目录下的打包文件名称 “`backup.tar.gz`”, 以及从 `app.arguments().at(1)` 中获得的待备份路径名。最后通过函数 `proc.start("tar", arguments)` 启动 `tar` 进程。

```
if (!proc.waitForStarted())
    return false;
proc.closeWriteChannel();
```

```

QByteArray procOutPut;
while (!proc.waitForFinished(0))
{
    if(proc.waitForReadyRead(10))
    {
        procOutPut = proc.readAll();
        qDebug()<<procOutPut;
    }
}
return EXIT_SUCCESS;
}

```

一旦 `waitForStarted()` 函数成功返回, `tar` 程序就开始正常运行了。由于不需要向这个新启动的进程输入任何数据, 程序首先关闭了用于输入的写通道。但又希望得到这个进程的输出信息, 因此启动了一个 `while` 循环, 一刻不停的检查子进程是否结束。如果这个进程尚未结束, 且有可供读取的信息, 就读取这段时间内子进程的全部输出信息并打印到控制台。使用的 `waitForFinished()` 函数和 `waitForReadyRead()` 函数功能分别是阻塞等待进程结束和阻塞等待有可供读取的数据, 所带参数均表示阻塞时间 (以毫秒为单位)。输出结果如图 17-1 所示。

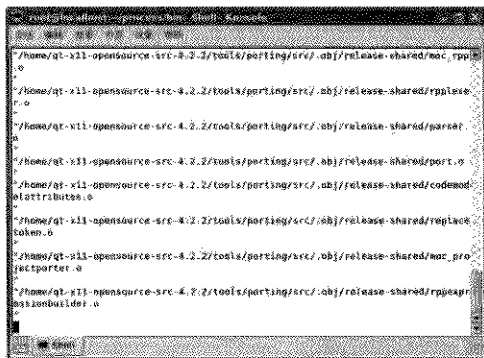


图 17-1 process 实例执行结果

最后, 为了操作方便, 还可以通过 `setEnvironment()` 函数和 `setWorkingDirectory()` 函数设置环境变量和工作路径, 默认的工作路径是进程被调用的当前路径。

17.2 Linux 进程间通信

Linux 下的进程包含以下几个关键要素:

- (1) 有一段可执行程序;
- (2) 有专用的系统堆栈空间;
- (3) 内核中有它的控制块 (进程控制块), 描述进程所占用的资源, 这样, 进程才能接受内核的调度;
- (4) 具有独立的存储空间。



进程的用户空间是互相独立的，一般是不能互相访问的。进程间通信广义上就是不同进程之间传播或交换信息，在这个意义上，两个进程通过磁盘上的文件交换信息，或者通过数据库中的某些表项或记录交互信息也可以称为是进程间通信。但常说的“进程间通信”是指有一定效率要求的通信手段，通常由操作系统内核提供支持，具体而言包括：消息队列、信号量、共享存储、SOCKET 和管道（包括普通管道、流管道和命名管道）。

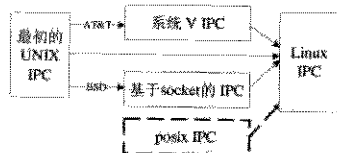


图 17-2 Linux 继承的进程间通信

Linux 下的进程通信手段基本上是从 UNIX 平台上的进程通信手段继承而来的。而对 UNIX 发展做出重大贡献的两大主力 AT&T 的贝尔实验室及 BSD（加州大学伯克利分校的伯克利软件发布中心）在进程间通信方面的侧重点有所不同。前者对 UNIX 早期的进程间通信手段进行了系统的改进和扩充，形成了“system V IPC”。通信进程局限在单个计算机内；后者则跳过了该限制，形成了基于套接字（socket）的进程间

通信机制。Linux 则把两者继承了下来，如图 17-2 所示。

其中，最初 UNIX IPC 包括：管道、FIFO、信号；System V IPC 包括：System V 消息队列、System V 信号灯、System V 共享内存区；Posix IPC 包括 Posix 消息队列、Posix 信号灯、Posix 共享内存区。有两点需要简单说明一下：

（1）由于 UNIX 版本的多样性，电子电气工程协会（IEEE）开发了一个独立的 UNIX 标准，这个新的 ANSI UNIX 标准被称为计算机环境的可移植性操作系统界面（Posix）。现有大部分 UNIX 和流行版本都是遵循 Posix 标准的，而 Linux 从一开始就遵循 Posix 标准。

（2）BSD 操作系统并不是没有涉及单机内的进程间通信（socket 本身就可以用于单机内的进程间通信）。事实上，很多 UNIX 版本的单机 IPC 留有 BSD 操作系统的痕迹，如 4.4BSD 支持的匿名内存映射、4.3+BSD 对可靠信号语义的实现等。

图 17-2 给出了 Linux 所支持的各种 IPC 手段。在接下来的讨论中，为了避免概念上的混淆，在尽可能少提及 UNIX 的各个版本的情况下，所有问题的讨论最终都会归结到 Linux 环境下的进程间通信。并且，对于 Linux 所支持通信手段的不同实现版本（如对于共享内存来说，有 Posix 共享内存区和 System V 共享内存区两个实现版本），将主要介绍 Posix API。

Linux 下进程间通信的几种主要手段包括：

（1）管道（pipe）及有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，除具有管道所具有的功能外，它还允许无亲缘关系进程间进行通信。

（2）信号（signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给自身；Linux 除了支持 UNIX 早期信号语义函数 `signal` 外，还支持语义符合 Posix.1 标准的信号函数“`sigaction()`”（实际上该函数是基于 BSD 系统的，BSD 系统为了实现在可靠信号机制，又能够统一对外接口，用 `sigaction()` 函数重新实现了 `signal()` 函数）。

（3）消息（message）队列：消息队列是消息的链接表，包括 Posix 消息队列和 System V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读取队列中的消息。消息队列克服了信号承载信息量少、管道只能承载无格式字节流及缓冲区大小受限等缺点。

（4）共享内存：多个进程可以访问同一块内存空间。它是针对其他通信机制运行效率较低而设计的，其最快的可用 IPC 形式。共享内存往往与其他通信机制，如信号量结合使用，来实现进程间的同步及互斥。

（5）信号量（semaphore）：主要作为进程间及同一进程不同线程之间的同步手段。

(6) 套接字(socket): 更为一般的进程间通信机制, 可用于不同机器之间的进程间通信。起初是由 UNIX 系统的 BSD 分支开发出来的, 但现在可以移植到其他类 UNIX 系统上, Linux 和 System V 的变种都支持套接字。

由于 Linux 进程间通信是一个庞大的话题, 这里限于篇幅没有详述每一种进程间通信手段的使用方法, 有兴趣的读者可以参考以下三本资料, 它们从不同角度各自做出了详细论述。

(1) 《UNIX 网络编程第二卷: 进程间通信》, 作者: W.Richard Stevens, 译者: 杨继张, 清华大学出版社。该书具有丰富的 UNIX 进程间通信实例及分析, 对 Linux 环境下的程序开发有极大的启发意义。

(2) 《UNIX 环境高级编程》, 作者: W.Richard Stevens, 译者: 尤晋元等, 机械工业出版社。该书具有丰富的编程实例, 以及对 UNIX 的发展历程中关键函数衍化的讲解。

(3) 《Linux 内核源代码情景分析(上、下)》, 作者: 毛德操、胡希明著, 浙江大学出版社。它是当读者在某些实现细节上产生疑问时最好的参考资料。

17.3 新型进程间通信——D-Bus

典型的桌面系统都会有多个应用程序在运行, 而且它们经常需要进行通信。DCOP (Desktop Communications Protocol) 是一个用于 KDE 的解决方案, 但是它依赖于 Qt, 所以不能用于其他桌面环境之中。类似地, Bonobo 是一个用于 GNOME 的解决方案, 由于它基于 CORBA (Common Object Request Broker Architecture) 的, 因此非常笨重。Bonobo 还依赖于 GObject, 所以也不能用于 GNOME 之外的桌面环境。D-Bus 的目标是将 DCOP 和 Bonobo 替换为简单的 IPC, 并集成这两种桌面环境。由于尽可能地减少了 D-Bus 所需的依赖, 所以其他使用 D-Bus 的应用程序不用担心需引入过多依赖。D-Bus 是一个大有前途的消息总线 and 活动系统, 正开始深入地渗透到 Linux 桌面之中。

17.3.1 D-Bus 简介

现有许多不同的 IPC 实现中每一种都定位于解决特定的问题。如 CORBA 是用于面向对象编程中复杂 IPC 的一个强大解决方案; DCOP 是一个较轻量级的 IPC 框架, 功能较少, 但是可以很好地集成到 KDE 桌面环境中; SOAP 和 XML-RPC 设计用于 Web 服务, 因而使用 HTTP 作为其传输协议。

D-Bus 是一种进程间通信 (IPC, Inter Process Communication) 和远程调用 (RPC, Remote Procedure Call) 机制, 最初为 Linux 系统开发, 它用一套统一的协议代替现有系统中互相竞争的 IPC 解决方案。它的设计同样支持系统级进程 (如打印机和硬件驱动服务) 和普通用户进程间的通信。D-Bus 是一种高速的二进制消息传输协议, 具有低延时和低开销的特点, 非常适合本机通信。它的详细规范目前由 freedesktop.org 项目制定, 大家均可免费获得。

典型的 D-Bus 设置由几个总线构成。一个持久的系统总线 (system bus), 它在引导时就会启动。这个总线由操作系统和后台进程使用, 安全性非常好, 任何应用程序都无法欺骗系统事件。还可以同时存在多个会话总线 (session buses), 这些会话总线在用户登录后启动, 属登录用户私有, 用于用户应用程序间的通信。当然, 如果一个应用程序需要接收来自系统总线的消息, 也可以直接连接到系统总线, 但是此时允许发送的消息将受到限制。通过总线通信时, 应用程序可以获取其他可用的对象和服务, 同时其自身也将成为一个活跃的被请求对象。总线适应于有多对多通信需求的场景, 但 D-Bus 同时也支持应用程序到应用程序的直接通信。下面我们将简单介绍一些 D-Bus 的基本概念, 它们是消息 (Messages)、服务名 (Service Names)、对象路径 (Object Paths) 和接口 (Interfaces)。



1. 消息 (Messages)

底层应用在 D-Bus 上的通信通过发消息进行。消息常被用于接力远程过程调用、应答及其所伴随的错误。使用总线时,消息有一个目的地址,它将被传递到感兴趣的部分,从而避免因“蜂群移动 (swarming)”或广播造成的拥塞。但有一种特殊的消息,称为“信号消息 (signal message)” (这个概念基于 Qt 的信号和槽机制)却没有预先定义的目的地址。因为它的目的是用在一对多时,信号消息被设计成工作在“选择 (opt-in)”机制下。

2. 服务名 (Service Names)

通过总线通信时,应用需要有一个服务名,用于在同一总线上被其他应用获取。服务名由 D-Bus 的守护进程代理,用于将消息从一个应用路由到另一个应用。一个类似服务名的概念是 IP 地址和主机名,通常计算机有一个 IP 地址并有一个或多个主机名,根据服务的不同供网络使用。另一方面,如果没有使用总线,就无须服务名了。这种情况类似在点对点网络中,由于端点已知,因此没有必要再使用主机名或 IP 地址来找到它。D-Bus 服务名的格式非常类似主机名,它是一系列由点分开的字母或数字。实际情况下通常根据定义服务的组织域名来命名服务名。例如, D-Bus 服务由 freedesktop.org 定义,那么它在总线上可以通过“org.freedesktop.DBus”服务名找到。

3. 对象路径 (Object Paths)

类似网络主机,一个应用通过暴露对象为其他应用提供特定服务。这些对象是层次组织的,非常像父子关系,类均从 QObject 派生。有一点不同的是,这里存在“根对象”的概念,所有对象有一个终极祖先。如果与 Web 服务类比,对象路径等同于 URL 路径部分。D-Bus 中对象路径的组织类似文件系统中的路径名,它是用“斜杠”分割的标签,每组标签包含字母、数字和下划线,它必须以“斜杠”开始但不必以“斜杠”结束。

4. 接口 (Interfaces)

接口类似于 C++ 的抽象类和 Java 中的 interface 关键字,它在调用者和被调用者间建立契约。也就是说,它们确定方法的名字、信号和可用属性以及通信建立后双方期望的行为。Qt 在插件系统中使用了一种非常类似的机制, C++ 基类通过 Q_DECLARE_INTERFACE() 宏关联了一个独一无二的标识。实际上, D-Bus 接口名的命名方式非常类似于 Qt 插件系统所建议的方式,一个标识通常由定义接口实体的域名构建。

为了方便记忆,总结上述概念如表 17-2 所示。

表 17-2 D-BUS 概念类比

D-BUS 概念	类 比	名字格式
服务名	网络主机名	点分隔
对象路径	URL 路径部分	斜杠分隔
接口	插件标识	点分隔

从 4.2 版本开始, Qt 提供了一个 QtDBus 模块,它使用 D-Bus 协议。应用程序可以通过该模块输出对象来为其他程序提供服务。其他程序可以像调用自身函数或访问自身对象属性一样访问这些远程对象。此外, QtDBus 模块还对信号和槽机制作了扩展,允许将远程信号和本地信号挂接到远程的槽中。接下来将介绍如何安装这个模块。

17.3.2 安装 QtDBus 模块

安装 QtDBus 模块需要 0.62 版本以上的 D-Bus 库。可以通过 rpm 命令查询当前系统中的 D-Bus 信息，如下所示。

```
[root@localhost Linux]# rpm -qa|grep dbus
dbus-glib-0.61-3
dbus-python-0.61-3
dbus-0.61-3
dbus-sharp-0.61-3
dbus-x11-0.61-3
dbus-devel-0.61-3
```

上述输出是在 Fedora 5 系统中的查询结果，因此必须安装一个较高的 D-Bus 版本。可以从 <http://www.freedesktop.org/wiki/Software/dbus> 站点下载最新的源代码包，当前发布的最新版本是 1.0.2。下面可以按如下步骤进行安装，且无须卸载原有版本。

```
[root@localhost dbus-1.0.2]# tar zxvf dbus-1.0.2.tar.gz
[root@localhost dbus-1.0.2]# cd dbus-1.0.2
[root@localhost dbus-1.0.2]# ./configure
```

配置成功后我们得到如下信息。

```
D-Bus 1.0.2
=====

prefix:                /usr/local
exec_prefix:            ${prefix}
libdir:                 /usr/local/lib
bindir:                 /usr/local/bin
sysconfdir:             /usr/local/etc
localstatedir:          /usr/local/var
datadir:                /usr/local/share
source code location:   .
compiler:               gcc
cflags:                 -g -O2 -Wall -Wchar-subscripts
                        -Wmissing-declarations -Wmissing-prototypes -Wnested-externs
                        -Wpointer-arith -Wcast-align -Wsign-compare
                        -Wdeclaration-after-statement -fno-common -fPIC

cppflags:
cxxflags:               -g -O2
64-bit int:             long long
32-bit int:             int
16-bit int:             short
Doxygen:                /usr/bin/doxygen
xmlto:                  /usr/bin/xmlto

Maintainer mode:        no
gcc coverage profiling: no
```



```

Building unit tests:      no
Building verbose mode:   no
Building assertions:     no
Building checks:         yes
Building SELinux support: yes
Building dnotify support: yes
Building X11 code:       yes
Building Doxygen docs:   yes
Building XML docs:       yes
Gettext libs (empty OK):
Using XML parser:        expat
Init scripts style:      redhat
Abstract socket names:   yes
System bus socket:       /usr/local/var/run/dbus/system_bus_socket
System bus address:      unix:path=/usr/local/var/run/dbus/system_
                        bus_socket
System bus PID file:      /usr/local/var/run/messagebus.pid
Session bus socket dir:   /tmp
Console auth dir:        /var/run/console/
Console owner file:      no
Console owner file path:
System bus user:         *      messagebus
Session bus services dir: /usr/local/share/dbus-1/services
'make check' socket dir: /tmp

```

接下来编译、安装即可，如下所示。

```

[root@localhost dbus-1.0.2]# make
[root@localhost dbus-1.0.2]# make install

```

D-Bus 1.0.2 版本安装成功后就可以安装 QtDBus 模块了。首先找到先前 Qt 源代码包展开后的目录进行配置，如下所示。

```

[root@localhost qt-x11-opensource-src-4.3.2]# ./configure -I
/usr/local/lib/dbus-1.0/include
-I /usr/local/include/dbus-1.0
-L /usr/local/lib
-ldbus-1
-qdbus
-v

```

其中 `-qdbus` 参数告诉 `configure` 命令强制包含 QtDBus 模块，如果配置成功将得到如下输出。

```

qmake switches ..
Build ..... libs tools examples
Configuration ..... release shared dll largefile stl separate_debug_info sse
qt3support accessibility minimal-config small-config medium-config large-config
full-config reduce_exports ipv6 getaddrinfo ipv6ifname getifaddrs inotify system-jpeg
system-mng system-png png no-gif system-freetype system-zlib nis iconv glib qdbus x11sm
xshape xinerama xcursor xfixes xrandr xrender fontconfig tablet xkb release

```

```

Debug..... no
Qt 3 compatibility... yes
QtDBus module..... yes
STL support ..... yes
PCH support ..... no
MMX/SSE support .... yes
IPv6 support ..... yes
IPv6 ifname support . yes
getaddrinfo support . yes
getifaddrs support... yes
Accessibility ..... yes
NIS support ..... yes
CUPS support ..... no
Iconv support ..... yes
Glib support ..... yes
Large File support .. yes
GIF support ..... no
JPEG support ..... plugin (system)
PNG support ..... yes (system)
MNG support ..... plugin (system)
zlib support ..... system
OpenGL support ..... no
NAS sound support .. no
Session management .. yes
XShape support ..... yes
Xinerama support .... yes
Xcursor support ..... yes
Xfixes support ..... yes
Xrandr support ..... yes
Xrender support ..... yes
FontConfig support .. yes
XKB Support ..... yes
imodule support .... yes
SQLite support ..... plugin (qt)

```

其中 QtDBus module 为 “yes”，接下来进行 Qt 的编译、安装即可，操作如下：

```

[root@localhost qt-x11-opensource-src-4.3.2]#make
[root@localhost qt-x11-opensource-src-4.3.2]#make install

```

如果用户操作系统中的 D-Bus 版本足够高，那么无须进行上述操作，因为 Qt 的 configure 命令将自动把 QtDBus 模块纳入其中。

17.3.3 接口与适配器

与大多数分布式应用一样，基于 D-Bus 的应用程序通常也包含客户端和服务端两部分。提供服务的对象实现通过适配器 (Adaptor) 在 D-Bus 上输出一个访问接口，客户端则使用这个标准接口找到该对象，然后使用服务对象的相关功能。如图 17-3 所示。

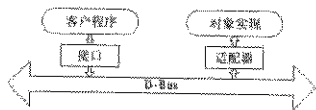


图 17-3 对象请求流程

接口是远程服务对象为客户程序提供的访问代理，客户程序可以利用接口调用远程对象的方法，连接远程对象的信号，以及使用 `get/set` 方法读写远程对象属性。在 `QtDBus` 模块中，接口有一个称为 `QDBusAbstractInterface` 的基类，所有接口都从它继承而来。但通常情况下，接口类的代码是由 `dbusxml2cpp` 工具自动生成的，无须手动编写（具体操作在下一节 `QtDBus` 应用实例中将详细介绍），这种方法通常也称为静态调用。`QtDBus` 模块还提供了一个实用的远程对象访问接口 `QDBusInterface`，当没有静态生成的接口时，它将提供动态的远程对象访问方式。例如，假设调用 `D-Bus` 上服务名、对象路径和接口名分别为 `com.example.Calculator`、`com.example.Calculator.org.mathematics.RPNCalculator` 的对象进行简单的“2¹⁰”数学运算，代码大致如下：

```

QDBusInterface remoteApp( "com.example.Calculator", "/Calculator/Operations",
                          "org.mathematics.RPNCalculator" );
remoteApp.call( "Push", 2 );
remoteApp.call( "Push", 2 );
remoteApp.call( "Execute", "^" );
QDBusReply<int> reply = remoteApp.call( "Pop" );
if ( reply.isValid() )
    printf( "%d", reply.value() );
  
```

计算过程是在客户端通过 `QDBusInterface` 接口调用 `call()` 函数，依次访问远程对象的压栈函数 `Push()`，计算函数 `Execute()` 和出栈函数 `Pop()`，最后打印返回结果，整个计算风格类似汇编程序。`QtDBus` 模块提供的另一个实用接口类是 `QDBusConnectionInterface`，用于访问 `D-Bus` 守护进程提供的服务，如访问当前连接到总线上的用户列表信息等。

在 `QtDBus` 模块中，适配器是一种特殊的类，它可以附着在任何从 `QObject` 继承而来的对象上，并为使用 `D-Bus` 的外部系统提供该对象的访问接口。适配器类只做了轻量级的简单封装，主要目的是在实体对象间传递调用，进行验证或转换外部输入格式，从而保护实体对象。与多继承不同，适配器允许在任意时刻加入对象，因此在输出现有对象时具有更大的灵活性。它的另一个好处是，可以在不同的接口中以相同的名字提供类似但不完全相同的功能，例如，为某个现有对象增加一个新版本的接口。

自定义适配器类必须从 `QDBusAbstractAdaptor` 继承，由于它也是 `QObject` 的子类，因此 `Q_OBJECT` 宏必须在声明中出现，以便源文件可以被 `moc` 工具处理。适配器还必须包含至少一个以“`D-Bus Interface`”命名的 `Q_CLASSINFO` 项，作为其声明输出的接口。类中的公共槽函数可以被各种调用类型的消息通过总线访问。类中的信号也将自动在 `D-Bus` 上接力传递。同样的，以 `Q_PROPERTY` 声明的属性将自动输出到 `D-Bus` 的属性接口。但是，由于 `QObject` 不允许存在非可读属性，因此，在适配器中也不能声明只写属性。

`QDBusAbstractAdaptor` 类是所有 `D-Bus` 适配器的基类，同时也是所有对象在 `D-Bus` 上对外输出标准接口的起点。在 `D-Bus` 上输出一个对象的步骤是：

- ❶ 从 `QDBusAbstractAdaptor` 继承得到一个具体的适配器。
- ❷ 将这个适配器附着到输出对象上，但要求该对象必须从 `QObject` 继承而来。
- ❸ 调用 `QDBusConnection::registerObject()` 注册这个输出对象。

为了更直观地理解适配器，这里自定义了一个适配器类 `MainAppAdaptor`，如下所示。

```

class MainAppAdaptor: public QDBusAbstractAdaptor
{
    Q_OBJECT
    Q_CLASSINFO("D-Bus Interface", "com.example.DBus.MainApp")
    Q_PROPERTY(QString caption READ caption WRITE setCaption)
private:
    QApplication *app;
public:
    MainAppAdaptor(QApplication *application)
        : QDBusAbstractAdaptor(application), app(application)
    {
        connect(application, SIGNAL(aboutToQuit()), SIGNAL(aboutToQuit()));
    }
    QString caption()
    {
        if (app->hasMainWindow())
            return app->mainWindow()->caption();
        return QString("");
    }
    void setCaption(const QString &newCaption)
    {
        if (app->hasMainWindow())
            app->mainWindow()->setCaption(newCaption);
    }
public slots:
    Q_ASYNC void quit()
    { app->quit(); }
signals:
    void aboutToQuit();
};

```

适配器 `MainAppAdaptor` 定义了一个 `com.example.DBus.MainApp` 接口，它包含读写属性 `caption`、方法 `quit()` 和信号 `aboutToQuit()`。其中，接口名称由 `Q_CLASSINFO` 宏定义，它包含约定的 D-Bus Interface 项；属性 `caption` 由 `PROPERTY` 宏定义，并提供了读写方法；信号的定义与普通 `QObject` 函数中的信号定义没有区别；方法 `quit()` 的定义与普通槽函数定义一样，但在返回值前有一个 `Q_ASYNC` 宏，它表明客户端以异步方式调用这个方法，即客户端通过 `QDBusAbstractInterface::call()` 函数调用远程的 `quit()` 方法时无须等待该方法执行完成就能返回。最后，在构造函数中将传入的 `QApplication` 对象的 `aboutToQuit()` 信号挂接到适配器中定义的 `aboutToQuit()` 信号，从而使得该信号可以在 D-Bus 上挂接远程槽函数。

利用这个适配器可以将一个 `QApplication` 对象输出到 D-Bus 上，如下所示。

```

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    new MainApplicationAdaptor(app);
    QDBus::sessionBus().registerService("org.freedesktop.MainApp");
    QDBus::sessionBus().registerObject("/MainApplication", app);
    .....
}

```



```

app.exec();
}

```

使用 `registerService()` 和 `registerObject()` 函数分别注册一个服务名和对象。需要注意的是, 从 `QDBusAbstractAdaptor` 继承来的类必须通过 `new` 操作符在堆中创建, 并且不能由用户删除, 它将在其附着的实际对象被删除后自动删除。

最后, D-Bus 对传输的消息类型有一定限制, `QtDBus` 模块对其进行了一些修改以适应 Qt 应用对数据类型的使用需求, 两者支持的原始类型如表 17-3 所示。

表 17-3 原始数据类型对照

Qt 类型	D-BUS 类型
uchar	BYTE
bool	BOOLEAN
short	INT16
ushort	UINT16
int	INT32
uint	UINT32
qlonglong	INT64
qulonglong	UINT64
double	DOUBLE
QString	STRING
QDBusVariant	VARIANT
QDBusObjectPath	OBJECT_PATH
QDBusSignature	SIGNATURE

此外, D-Bus 允许通过 `ARRAY`、`STRUCT` 和 `maps/dictionaries` 三种方式定义原始数据类型的复合。在 `QtDBus` 模块中, 用户定义自己的数据类型还必须通过 `Q_DECLARE_METATYPE()` 宏进行声明, 然后调用 `qDBusRegisterMetaType()` 函数进行注册。

17.3.4 QtDBus 应用实例

这一节将使用 `QtDBus` 模块编写一个 D-Bus 应用程序, 这个程序由客户端和服务端两部分组成, 实现了住房贷款计算功能, 如图 17-4 所示。

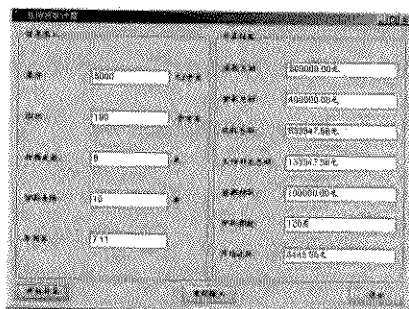


图 17-4 住房贷款计算实例

首先来看服务器端的程序,它通过 `Calculate` 类完成贷款业务的各项计算功能。`Calculate` 类定义如下:

```
class Calculate : public QObject
{
    Q_OBJECT
public:
    Calculate();
public Q_SLOTS:
    double housePrice();
    double lead();
    double payPerMonth();
    int totalMonths();
    double initPay();
    double totalPay();
    double totalInterest();
    void initInput(double price,double area,int percent,double interest,
                  int year);
private:
    double m_price;
    double m_area;
    int m_percent;
    double m_interest;
    int m_year;
};
```

`Calculate` 类通过 `initInput()` 函数输入基本信息,其他槽函数完成对应贷款业务功能的计算。

```
void Calculate::initInput(double price,double area,
                          int percent,double interest,int year)
{
    m_price = price;
    m_area = area;
    m_percent = percent;
    m_interest = interest;
    m_year = year;
}
```

函数 `initInput()` 输入住房单价 `price`、住房面积 `area`、按揭成数 `percent`、贷款利率 `interest` 和贷款年限 `year` 五项基本信息,并保存在各自私有变量中供其他函数使用。

```
double Calculate::housePrice()
{
    double houseprice = m_price*m_area;
    return houseprice;
}
```

函数 `housePrice()` 计算住房总价值并将其返回,其中“住房总价值=住房单价×住房面积”。

```
double Calculate::lead()
{
    double load = housePrice()*m_percent*0.1;
```



```

    return lead;
}

```

函数 `lead()` 计算贷款总额并将其返回，其中“贷款总额=住房总价值×按揭成数×0.1”，这里需要把按揭成数转换成百分比。

```

double Calculate::payPerMonth()
{
    double capital = m_price*m_area*m_percent*0.1;
    double monthinterest = m_interest/12*0.85*0.01;
    double temp = pow((1+monthinterest), (m_year*12));
    double paypermonth = (capital*monthinterest*temp)/(temp-1);
    return paypermonth;
}

```

函数 `payPerMonth()` 计算每月还款数并将其返回。每月还款数按公式 $A \times I \times (1+I)^T / [(1+I)^T - 1]$ 计算，其中 A 为贷款总额， I 为月利率， T 为贷款期数。在使用该公式时，需要将输入的年利率除以 12 转换成月利率，另外，首次贷款还可享受 15% 的利率优惠；贷款期数由输入的贷款年限乘以 12 得到。

```

int Calculate::totalMonths()
{
    int totalmonths = m_year*12;
    return totalmonths;
}

```

函数 `totalMonths()` 十分简单，它返回贷款的月数。

```

double Calculate::initPay()
{
    double initpay = housePrice()*(1-m_percent*0.1);
    return initpay;
}

```

函数 `initPay()` 计算首付款并将其返回。其中“首付款=住房总价值×(1-按揭成数×0.1)”。

```

double Calculate::totalPay()
{
    double totalpay = payPerMonth()*totalMonths();
    return totalpay;
}

```

函数 `totalPay()` 计算还款总额并将其返回。其中“还款总额=月还款数×还款期数”。

```

double Calculate::totalInterest()
{
    double totalinterest = totalPay()-lead();
    return totalinterest;
}

```

函数 `totalInterest()` 计算产生的利息总额并将其返回。其中“利息总额=还款总额-贷款总额”。

功能类编写完成后需要为其编写一个适配器类，用于将 `Calculate` 对象输出到 D-Bus 上。这里使用 `QtDBus` XML 编译器自动生成一个适配器。首先，从功能类 `Calculate` 产生一个用于接口描述的 XML

文件 `calculate.xml`，如下所示：

```
[root@stephen temp]# gdbuscpp2xml calculate.h -o calculate.xml
```

生成的 `calculate.xml` 内容如下：

```
<node >
  <interface name="org.freedesktop.Qt.CalculateInterface">
    <method name="payPerMonth">
      <arg type="d" direction="out"/>
    </method>
    <method name="totalMonths">
      <arg type="i" direction="out"/>
    </method>
    <method name="initPay">
      <arg type="d" direction="out"/>
    </method>
    <method name="totalInterest">
      <arg type="d" direction="out"/>
    </method>
    <method name="totalPay">
      <arg type="d" direction="out"/>
    </method>
    <method name="lead">
      <arg type="d" direction="out"/>
    </method>
    <method name="housePrice">
      <arg type="d" direction="out"/>
    </method>
    <method name="initInput">
      <arg name="price" type="d" direction="in"/>
      <arg name="area" type="d" direction="in"/>
      <arg name="percent" type="i" direction="in"/>
      <arg name="interest" type="d" direction="in"/>
      <arg name="year" type="i" direction="in"/>
    </method>
  </interface>
</node>
```

这个文件定义了一个接口名“`org.freedesktop.Qt.CalculateInterface`”和 8 个方法，每个方法所带参数由 `arg` 标签逐一列出，如表 17-4 所示。

表 17-4 参数描述表

name	type	direction
参数名	参数类型	输入或输出参数，由 in、out 或 inout 表示

其中参数类型如表 17-5 所示。



表 17-5 参数类型

名 称	代 码	描 述
INVALID	0 (ASCII NUL)	无效类型
BYTE	121 (ASCII 'y')	8-bit 无符号整型
BOOLEAN	98 (ASCII 'b')	布尔值, 0 为假, 1 为真, 其他值无效
INT16	110 (ASCII 'n')	16-bit 有符号整型
UINT16	113 (ASCII 'q')	16-bit 无符号整型
INT32	105 (ASCII 't')	32-bit 有符号整型
UINT32	117 (ASCII 'u')	32-bit 无符号整型
INT64	120 (ASCII 'x')	64-bit 有符号整型
UINT64	116 (ASCII 'r')	64-bit 无符号整型
DOUBLE	100 (ASCII 'd')	IEEE 754 double
STRING	115 (ASCII 's')	UTF-8 string (must be valid UTF-8), 必须以 null 结尾, 同时不能包含其他 null 字节
OBJECT_PATH	111 (ASCII 'o')	对象实例名
SIGNATURE	103 (ASCII 'g')	A type signature
ARRAY	97 (ASCII 'a')	Array
STRUCT	114 (ASCII 'r'), 40 (ASCII 'i'), 41 (ASCII 'j')	Struct
VARIANT	118 (ASCII 'v')	Variant 类型
DICT_ENTRY	101 (ASCII 'e'), 123 (ASCII 'l'), 125 (ASCII 't')	dict 或 map (array of key-value pairs)中的一项

这个描述接口的 XML 文件非常类似 CORBA 的接口定义语言 IDL。在确认该 XML 文件的内容无误后将通过它生成相应的接口和适配器类, 否则可以对这个文件做适当的手动修改。可以通过如下命令生成适配器类。

```
[root@stephen temp]# qdbusxml2cpp calculate.xml -a calculate_adaptor
```

所产生的适配器代码如下所示。

```
class CalculateAdaptor: public QDBusAbstractAdaptor
{
    Q_OBJECT
    Q_CLASSINFO("D-Bus Interface", "org.freedesktop.Qt.CalculateInterface")
    Q_CLASSINFO("D-Bus Introspection", "")
    "<interface name=\"org.freedesktop.Qt.CalculateInterface\" >\n"
    "<method name=\"payPerMonth\" >\n"
    "<arg direction=\"out\" type=\"d\" />\n"
    "</method>\n"
    "<method name=\"totalMonths\" >\n"
    "<arg direction=\"out\" type=\"i\" />\n"
    "</method>\n"
    "<method name=\"initPay\" >\n"
    "<arg direction=\"out\" type=\"d\" />\n"
    "</method>\n"
    "<method name=\"totalInterest\" >\n"
    "<arg direction=\"out\" type=\"d\" />\n"
```

```

"    </method>\n"
"    <method name=\"totalPay\" >\n"
"        <arg direction=\"out\" type=\"d\" />\n"
"    </method>\n"
"    <method name=\"lead\" >\n"
"        <arg direction=\"out\" type=\"d\" />\n"
"    </method>\n"
"    <method name=\"housePrice\" >\n"
"        <arg direction=\"out\" type=\"d\" />\n"
"    </method>\n"
"    <method name=\"initInput\" >\n"
"        <arg direction=\"in\" type=\"d\" name=\"price\" />\n"
"        <arg direction=\"in\" type=\"d\" name=\"area\" />\n"
"        <arg direction=\"in\" type=\"i\" name=\"percent\" />\n"
"        <arg direction=\"in\" type=\"d\" name=\"interest\" />\n"
"        <arg direction=\"in\" type=\"i\" name=\"year\" />\n"
"    </method>\n"
" </interface>\n"
" )
public:
    CalculateAdaptor(QObject *parent);
    virtual ~CalculateAdaptor();

public: // PROPERTIES
public Q_SLOTS: // METHODS
    double housePrice();
    void initInput(double price, double area, int percent, double interest, int year);
    double initPay();
    double lead();
    double payPerMonth();
    double totalInterest();
    int totalMonths();
    double totalPay();
Q_SIGNALS: // SIGNALS
};

```

关于适配器代码在上一节已做了详细介绍，这里不再重复了。需要注意的是，产生的适配器类如有需要，可以手动修改。最后在 main() 函数中完成注册即可，如下所示：

```

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    Calculate *calculate = new Calculate();
    new CalculateAdaptor(calculate);
    QDBusConnection connection = QDBusConnection::sessionBus();
    connection.registerObject("/Calculate", calculate);
    connection.registerService("org.freedesktop.DbusExample");
}

```



```

return app.exec();
}

```

服务器端编写完成后，可以首先让它执行起来。我们通过 Qt 4.3 安装目录下的 qdbusviewer 程序查看当前会话总线的内容，如图 17-5 所示，其中阴影部分为我们新加入的 Calculate 服务。

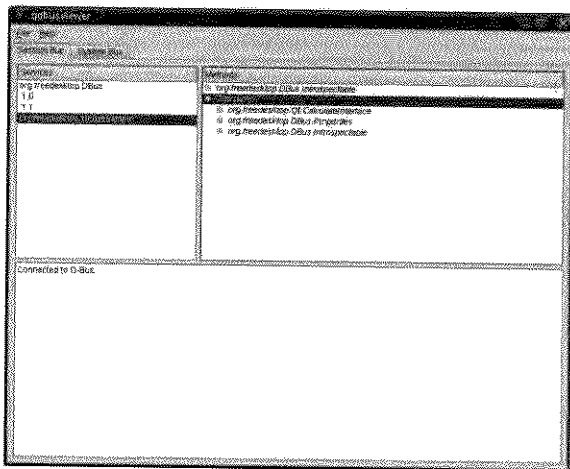


图 17-5 会话总线当前内容

在客户端，还是利用 calculate.xml 文件来生成所需的接口类，操作如下：

```
[root@stephen temp]# qdbusxml2cpp calculate.xml -p calculate_interface
```

它将产生如下接口代码：

```

class CalculateInterface: public QDBusAbstractInterface
{
    Q_OBJECT
public:
    static inline const char *staticInterfaceName()
    { return "org.freedesktop.Qt.CalculateInterface"; }
public:
    CalculateInterface(const QString &service, const QString &path,
        const QDBusConnection &connection, QObject *parent = 0);
    ~CalculateInterface();
public Q_SLOTS: // METHODS
    inline QDBusReply<double> housePrice()
    {
        QList<QVariant> argumentList;
        return callWithArgumentList(QDBus::Block,
            QLatin1String("housePrice"), argumentList);
    }
    inline QDBusReply<void> initInput(double price, double area,

```

```

        int percent, double interest, int year)
    {
        QList<QVariant> argumentList;
        argumentList << QVariantFromValue(price) << QVariantFromValue(area)
            << QVariantFromValue(percent)
            << QVariantFromValue(interest)
            << QVariantFromValue(year);
        return callWithArgumentList(QDBus::Block,
            QLatin1String("initInput"), argumentList);
    }

    inline QDBusReply<double> initPay()
    {
        QList<QVariant> argumentList;
        Return    callWithArgumentList(QDBus::Block,
            QLatin1String("initPay"), argumentList);
    }

    inline QDBusReply<double> lead()
    {
        QList<QVariant> argumentList;
        return callWithArgumentList(QDBus::Block, QLatin1String("lead"),
            argumentList);
    }

    inline QDBusReply<double> payPerMonth()
    {
        QList<QVariant> argumentList;
        return callWithArgumentList(QDBus::Block,
            QLatin1String("payPerMonth"), argumentList);
    }

    inline QDBusReply<double> totalInterest()
    {
        QList<QVariant> argumentList;
        return callWithArgumentList(QDBus::Block,
            QLatin1String("totalInterest"), argumentList);
    }

    inline QDBusReply<int> totalMonths()
    {
        QList<QVariant> argumentList;
        return callWithArgumentList(QDBus::Block,
            QLatin1String("totalMonths"), argumentList);
    }

    inline QDBusReply<double> totalPay()
    {
        QList<QVariant> argumentList;
        return callWithArgumentList(QDBus::Block,
            QLatin1String("totalPay"), argumentList);
    }
}

Q_SIGNALS: // SIGNALS
};

```



```
namespace org {
    namespace freedesktop {
        namespace Qt {
            typedef ::CalculateInterface CalculateInterface;
        }
    }
}
```

生成的代码如有需要,可以进行手动修改。接口类所做的工作就是使用 `callWithArgumentList()` 函数进行远程方法的调用。

```
class LoadForm: public QWidget, Ui::ShowWidget
{
    Q_OBJECT
public:
    LoadForm(QWidget *parent = 0);
    ~LoadForm();
private slots:
    void submitClickedSlot();
    void revertClickedSlot();
private:
    double price;
    double area;
    int percent;
    double interest;
    int year;
    CalculateInterface *calc;
};
```

界面显示类 `LoadForm` 获取用户输入,然后使用接口类 `CalculateInterface` 调用远程的方法,并最终在界面上输出返回结果。

```
LoadForm::LoadForm(QWidget *parent)
    : QWidget(parent)
{
    setupUi(this);
    calc = new CalculateInterface("org.freedesktop.DbusExample", "/Calculate",
                                   QDBusConnection::sessionBus(), this);
    connect(submitButton, SIGNAL(clicked(bool)), this,
            SLOT(submit ClickedSlot()));
    connect(revertButton, SIGNAL(clicked(bool)), this,
            SLOT(revert ClickedSlot()));
}
```

构造函数 `LoadForm()` 用于生成界面,然后使用服务器端约定的服务名、对象和接口创建一个本地调用的接口对象 `calc`,并将“开始计算”和“重新输入”按钮关联到各自的槽函数。

```
void LoadForm::submitClickedSlot()
{
```



```

price = priceEdit->text().trimmed().toDouble();
area = areaEdit->text().trimmed().toDouble();
percent = percentEdit->text().trimmed().toInt();
interest = interestEdit->text().trimmed().toDouble();
year = yearEdit->text().trimmed().toInt();
calc->initInput(price, area, percent, interest, year);
housePriceEdit->setText(QString::tr("%1 元")
    .arg(calc->housePrice(), 0, 'f', 2));
leadEdit->setText(QString::tr("%1 元")
    .arg(calc->lead(), 0, 'f', 2));
totalPayEdit->setText(QString::tr("%1 元")
    .arg(calc->totalPay(), 0, 'f', 2));
totalInterestEdit->setText(QString::tr("%1 元")
    .arg(calc->totalInterest(), 0, 'f', 2));
initPayEdit->setText(QString::tr("%1 元")
    .arg(calc->initPay(), 0, 'f', 2));
totalMonthsEdit->setText(QString::tr("%1 月")
    .arg(calc->totalMonths()));
payEdit->setText(QString::tr("%1 元")
    .arg(calc->payPerMonth(), 0, 'f', 2));
}

```

当用户完成输入，按下“开始计算”按钮后，槽函数 `submitClickedSlot()` 开始执行。它首先获得用户输入（这里并没有进行输入信息的合法性检查，读者可以自行补全）。然后通过接口 `calc` 进行了一系列远程方法调用，并将返回值在界面中输出。在输出结果时，使用了 `QString` 的 `arg` 方法，用于保证小数点后两位有效数字的输出。

```

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForName("gb2312"));
    LoadForm loadForm;
    loadForm.show();
    return app.exec();
}

```

`main()` 函数的功能十分简单，它只需生成并显示输入界面类 `LoadForm` 即可。在客户端，省略了所有与界面相关的代码介绍，读者可以查看本章的实例代码。

17.4 小 结

本章介绍了进程和进程间通信的知识，首先介绍的是常见的进程起停和 Linux 进程间通信的基本知识，然后重点介绍了 Qt 中桌面环境下基于 D-Bus 的多进程应用程序开发，并以一个示例详细介绍了使用 Qt 的 `QtDBus` 模块开发 D-Bus 应用程序的基本流程。

第 18 章 Qt 插件

Qt 只是一个基础平台，并没有包罗万象，在需要对 Qt 自身或 Qt 应用程序进行扩展时，可以使用 Qt 的插件系统来完成所需的扩展功能。Qt 提供了两种 API 来创建插件：一种是扩展 Qt 本身的高级 API，如数据库插件、图像插件、风格插件等；另外一种扩展 Qt 应用程序的低级 API。低级别的插件可以是任何形式，而不仅限于数据库、风格和图像等领域。

18.1 Qt 插件开发基础

使用插件首先是定义统一的接口，然后还必须要解决两个问题，一个是在应用程序中如何使用插件，另外一个是如何编写插件。在 Qt 中，应用程序使用插件的步骤如下：

- ❶ 定义插件的接口（C++中，接口由只有纯虚函数的抽象类来实现）；
- ❷ 使用 `Q_DECLARE_INTERFACE()` 宏在 Qt 的元对象系统中注册接口信息；
- ❸ 在应用程序中使用 `QPluginLoader` 类装入插件；
- ❹ 使用 `qobject_cast()` 测试插件是否实现了指定的接口。

编写插件的步骤如下：

- ❶ 定义从 `QObject` 以及需要实现的接口继承的插件类；
- ❷ 使用 `Q_INTERFACE()` 宏在 Qt 的元对象系统中注册接口信息；
- ❸ 使用 `Q_EXPORT_PLUGIN2()` 宏导出插件；
- ❹ 建立 `qmake` 文件并构建插件。

如下代码定义了一个插件接口类。

```
class TacticsInterface
{
public:
    virtual ~TacticsInterface() {}
    virtual int calculate(int x, int y) = 0;
}
Q_DECLARE_INTERFACE(TacticsInterface, "com.zeki.tactics/1.0")
```

在该接口类中，定义了一个纯虚函数 `calculate()`，该函数是插件显露给应用程序的接口函数。`Q_DECLARE_INTERFACE()` 类的作用是注册接口，接下来定义实现类 `TacticsPlugin`。

```
class TacticsPlugin : public QObject, public TacticsInterface
{
    Q_OBJECT
    Q_INTERFACES(TacticsInterface)
public:
    int calculate(int x, int y);
};
```

在实现类 `TacticsPlugin` 中, 使用了 `Q_INTERFACES` 宏在 Qt 的元对象系统中注册了该接口。接口定义后, 就可以具体实现相应的功能, 这将在后面展开。

和普通应用程序相比, 插件的调试比较麻烦。为了简化开发过程, 可以先开发类似的独立程序, 再改为插件, 这样可以减少调试工作量。使用插件时还可以将环境变量 `QT_DEBUG_PLUGIN` 设为非零值, 这样就可以输出一些插件调试信息。

18.2 Qt 设计器插件

Qt 设计器提供了丰富的窗口部件, 但在实际应用中, 这些通用的窗口部件不一定能满足特定领域的需求。Qt 设计器的插件扩展机制正是为了满足这一需求而产生的。

在介绍 Qt 设计器的插件机制之前, 首先介绍比较简单的复用或扩展 Qt 设计器窗口部件的方法。

18.2.1 使用 Scratchpad

在使用 Qt 设计器时, 可能有一些窗口部件的大部分属性是相同的, 对于这种情况, 可以进行复制来节省工作量。但复制只能在当前会话中使用, 而且一次只能复制一种类型的窗口部件。这时可以使用 Qt 的 `Scratchpad` 将这些具有公共特性的窗口部件保存起来, 供下次使用。下面通过实例来讲解操作过程。

首先, 新建一个基于 `Dialog` 的 `ui` 文件, 并在对话框上放置一个按钮 (`Push Button`)。接下来将按钮的 `objectName` 属性改为 `MyPushButton`, 并将该按钮拖动到“`Widget Box`”窗口上, 可以看到在“`Widget Box`”上新增了一栏“`Scratchpad`”, 在该栏里有创建的 `MyPushButton` 按钮, 如图 18-1 所示。

`Scratchpad` 中的窗口部件由 Qt 设计器管理, 每次进行 `ui` 设计时均可使用。如果需要改变窗口部件的名称, 单击鼠标右键选择“`Edit Name`”菜单项进行修改, 删除则选择“`Remove`”菜单项。

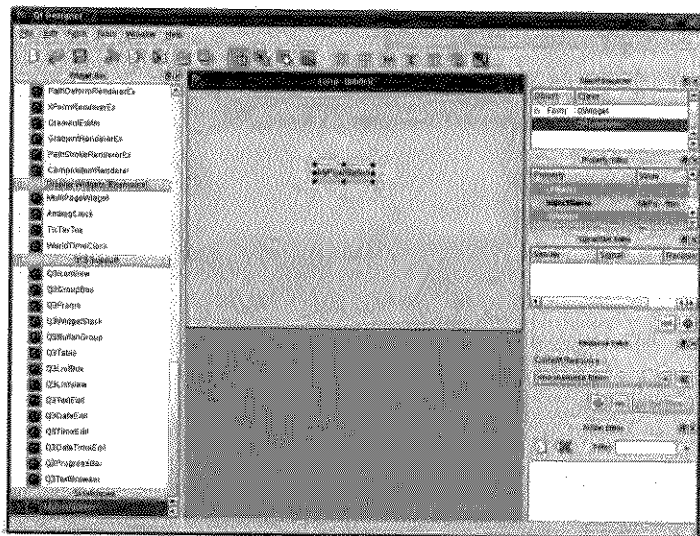


图 18-1 Scratchpad

18.2.2 提升自定义窗口部件

上一节的 Scratchpad 只是实现了窗口部件复用（其实是复制），并未对窗口部件进行扩展。如果需要使用自定义的窗口部件则需要自己定义窗口部件的各种行为，并在 Qt 设计器中指明具体的窗口部件使用该类。

例如想在一个 QLabel 的窗口部件中完成自定义的绘图操作，可以定义类如下：

```
class MyLabel : public QLabel
{
    Q_OBJECT

public:
    MyLabel(QWidget *parent = 0);

protected:
    void paintEvent(QPaintEvent *event);
}
```

将 MyLabel 类的声明放在头文件 mylabel.h 中，实现保存在 mylabel.cpp 中。

接下来在 Qt 设计器中新建一个 ui 文件，并在 Dialog 或 Widget 上放置一个 Label 窗口部件，在 Label 窗口部件上单击鼠标右键，选择“Promote to Custom Widget”，出现对话框。在对话框中的“promoted class name”中输入 MyLabel，“Header file”中输入头文件名，这样该 Label 在使用 uic 生成源代码时，将是基于 MyLabel 类生成的对象。如图 18-2 所示。

使用文本编辑器打开 ui 文件，可以发现该 Label 在 ui 文件中表述为：

```
<widget class="MyLabel" name="label">
```

表明该 Label 是基于 MyLabel 类的窗口部件。

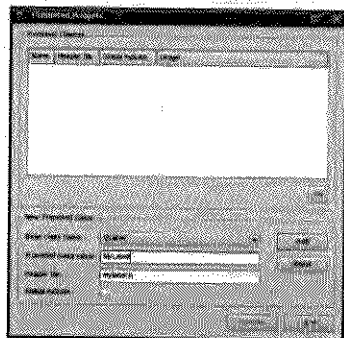


图 18-2 Promoted Widgets 对话框

18.2.3 Qt 设计器插件开发

实现功能最强的 Qt 窗口部件复用方式是使用 Qt 设计器的插件开发框架。在 Qt 设计器中划分了 4 种类型的扩展。分别是：

- 任务菜单扩展

任务菜单扩展基于 `QDesignerTaskMenuExtension` 类，完成扩展 Qt 设计器的右键菜单。使用该扩展在 Qt 设计器的任务菜单中加入自定义的菜单项。

- 容器扩展

容器扩展基于 `QDesignerContainerExtension` 类，该扩展可以实现多页容器插件，实现页的添加和删除。

- 成员表单扩展

成员表单扩展基于 `QDesignerMemberSheetExtension` 类，该扩展操作窗口部件的成员函数并显示信号和槽的连接。

- 属性页扩展

属性页扩展基于 `QDesignerPropertySheetExtension` 类，该类可以操作在 Qt 设计器的属性编辑器中出现的窗口部件属性。

为了创建扩展，必须从 `QObject` 和相应的基类继承并实现相应的函数。为了使 Qt 的元对象系统能够感知实现的接口，要使用 `Q_INTERFACES()` 宏，如下所示。

```
class MyExtension: public QObject,
                  public QDesignerContainerExtension
{
    Q_OBJECT
    Q_INTERFACES(QDesignerContainerExtension)
    ...
}
```

这样就可以使 Qt 设计器使用 `qobject_cast()` 函数来查询支持的接口。

下面通过建立一个指南针插件来说明 Qt 设计器插件的开发。该插件实现了一个类似于飞行模拟中常用的指南针，当飞机机头改变方向时，指南针也同时变换方向。

插件头文件定义如下：

```
#ifndef COMPASSPLUGIN_H
#define COMPASSPLUGIN_H

#include <QDesignerCustomWidgetInterface>

class CompassPlugin : public QObject,
                      public QDesignerCustomWidgetInterface
{
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetInterface)

public:
    CompassPlugin(QObject *parent = 0);

    bool isContainer() const;
    bool isInitialized() const;
    QIcon icon() const;
    QString domXml() const;
```



```
QString group() const;
QString includeFile() const;
QString name() const;
QString toolTip() const;
QString whatsThis() const;
QWidget *createWidget(QWidget *parent);
void initialize(QDesignerFormEditorInterface *core);

private:
    bool initialized;
};
#endif
```

这个头文件基本上是模板文件，大多数插件都大同小异。插件从 `QDesignerCustom WidgetInterface` 类继承，具体实现如下所示：

```
#include "compass.h"
#include "compassplugin.h"
#include <QtPlugin>

CompassPlugin::CompassPlugin(QObject *parent)
    : QObject(parent)
{
    initialized = false;
}

void CompassPlugin::initialize(QDesignerFormEditorInterface * /* core */)
{
    if (initialized)
        return;

    initialized = true;
}

bool CompassPlugin::isInitialized() const
{
    return initialized;
}

QWidget *CompassPlugin::createWidget(QWidget *parent)
{
    return new Compass(parent);
}

QString CompassPlugin::name() const
{
    return "Compass";
}
```

```

QString CompassPlugin::group() const
{
    return "My Widgets";
}

QIcon CompassPlugin::icon() const
{
    return QIcon();
}

QString CompassPlugin::toolTip() const
{
    return "Compass";
}

QString CompassPlugin::whatsThis() const
{
    return "Compass";
}

bool CompassPlugin::isContainer() const
{
    return false;
}

QString CompassPlugin::domXml() const
{
    return "<widget class=\"Compass\" name=\"Compass\">\n"
        "  <property name=\"geometry\">\n"
        "    <rect>\n"
        "      <x>0</x>\n"
        "      <y>0</y>\n"
        "      <width>100</width>\n"
        "      <height>100</height>\n"
        "    </rect>\n"
        "  </property>\n"
        "</widget>\n";
}

QString CompassPlugin::includeFile() const
{
    return "compass.h";
}

```

Q_EXPORT_PLUGIN2(compassplugin, CompassPlugin)

在函数 `createWidget()` 中，指明了如何创建插件实例。`name()` 函数返回插件名称，`group()` 指定



了插件在 Qt 设计器中所处的组，icon()返回插件的图标，toolTip()返回插件的工具提示。

插件的真正实现的头文件如下：

```
#ifndef COMPASS_H
#define COMPASS_H

#include <QTime>
#include <QWidget>
#include <QtDesigner/QDesignerExportWidget>

class QDESIGNER_WIDGET_EXPORT Compass : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(double angle READ angle WRITE setAngle)
    Q_PROPERTY(double second READ second WRITE setSecond)

public:
    Compass(QWidget *parent = 0);
    inline qreal angle()
    {
        return m_angle;
    };

    inline void setAngle(qreal angle)
    {
        m_angle = angle;
    };

    inline int second()
    {
        return m_second;
    };

    void setSecond(int second);

public slots:
    void setValue(qreal heading);

signals:
    void valueChanged(qreal heading);

protected:
    void paintEvent(QPaintEvent *event);

protected slots:
    void timerEvent(QTimerEvent *event);

private:
```



```

    qreal m_angle;
    qreal m_step;
    qreal m_animateAngle;
    int m_second;
};
#endif

```

在这里定义了插件的两个属性 `angle` 和 `second`，一个信号 `valueChanged()` 和一个槽函数 `setValue()`。指南针的实现如下：

```

#include <QtGui>
#include "compass.h"

const int labelX = -8;
const int labelY = -65;

Compass::Compass(QWidget *parent)
    : QWidget(parent)
{
    setWindowTitle(tr("Compass"));
    m_angle = 0; // North
    m_step = 0;
    m_animateAngle = 0;
    m_second = 0;
    resize(200, 200);
}
// 绘制指南针
void Compass::paintEvent(QPaintEvent *)
{
    static const QPoint needle[3] = {
        QPoint(7, 8),
        QPoint(-7, 8),
        QPoint(0, -70)
    };

    QColor poleColor(127, 0, 127);
    QColor scaleColor(0, 127, 127, 191);

    int side = qMin(width(), height());

    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing);
    painter.translate(width() / 2, height() / 2);
    painter.scale(side / 200.0, side / 200.0);

    painter.setPen(Qt::NoPen);
    painter.setBrush(poleColor);

```



```
painter.save();
painter.drawConvexPolygon(needle, 3);
painter.restore();

painter.setPen(poleColor);
QFont font;
font.setBold(true);
font.setPointSize(18);
painter.setFont(font);

painter.rotate(-m_animateAngle);
for (int i = 0; i < 8; ++i) {
    painter.drawLine(0, -96, 0, -88);
    switch(i) {
        case 0:
            painter.drawText(labelX, labelY, "N");
            break;
        case 2:
            painter.drawText(labelX, labelY, "E");
            break;
        case 4:
            painter.drawText(labelX, labelY, "S");
            break;
        case 6:
            painter.drawText(labelX, labelY, "W");
            break;
    }
    painter.rotate(45.0);
}

painter.setPen(scaleColor);

for (int j = 0; j < 60; ++j) {
    if(j % 15 != 0)
        painter.drawLine(92, 0, 96, 0);
    painter.rotate(6.0);
}
}

void Compass::setValue(qreal heading)
{
    m_angle = heading;
}

// 定时显示指南针偏转事件
void Compass::timerEvent(QTimerEvent *event)
{
    m_animateAngle += m_step;
    if(m_animateAngle <= m_angle)
```

```

        update();
    else
        killTimer(event->timerId());
}
// 设置动画事件
void Compass::setSecond(int second)
{
    m_second = second;
    m_animateAngle = 0;
    if (m_second <= 0)
        m_step = m_angle;
    else
        m_step = m_angle / (second * 20);
    startTimer(50);
}

```

最后要写 .pro 文件，其中包含：

```

TEMPLATE    = lib
CONFIG      += designer plugin debug_and_release

```

CONFIG 变量中的 designer 表示工程要和 libQtDesigner.so (或 QtDesigner4.dll) 库进行动态链接。plugin 说明工程生成的目标为插件库。

如果编译为既有 debug 又有 release 模式，在 Windows 或 Mac 系统上还需要为 debug 版目标执行文件加入合适的后缀，定义如下：

```

CONFIG(debug, debug|release) {
    mac: TARGET = $$join(TARGET,,,_debug)
    win32: TARGET = $$join(TARGET,,,d)
}

```

不过因为 Qt 设计器是 release 版，所以要编译 release 版的插件才能正确地 被 Qt 设计器加载。

为了将编译好的库放到 designer 的插件目录，让 Qt 设计器自动加载，在工程中还需要加入：

```

target.path = $$[QT_INSTALL_PLUGINS]/designer
INSTALLS += target

```

最后运行 make install 安装指南针插件，插件在 Qt 设计器中显示如图 18-3 所示。

在 Qt 设计器中可以试着改变指南针的 angle 和 second 属性看有什么变化。

如果希望插件能在 Qt 的 Visual Studio 集成版中出现，需要将 release 版的库文件复制到 Visual Studio 集成环境的 plugins 目录下。

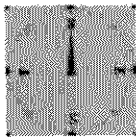


图 18-3 指南针插件

18.3 编写数据库插件

开源版的 Qt 支持 Borland InterBase、MySQL、ODBC、PostgreSQL、SQLite 等数据库接口，商业版的还支持 Sybase Adaptive Server、DB2 和 Oracle 数据库。如果使用 Qt 目前还不支持的数据库，可



以自己编写数据库驱动插件，由 Qt 动态加载。当然也可以直接使用数据库的 API 访问。本节介绍编写自定义的数据库驱动插件的过程。

在 Qt 的 SQL 模块中，QSqlDatabase 类负责装入和管理数据库驱动插件。当使用 QSqlDatabase::addDatabase() 打开数据库连接时，Qt 负责装入相应的数据库驱动插件。QSqlDatabase 依赖于驱动插件的 QSqlDriver 和 QSqlResult 接口完成相应的操作。

QSqlDriver 类是定义 SQL 数据库驱动的抽象基类。QSqlDriver 负责连接数据库，建立适当的环境，使用相应数据库的 API 创建 QSqlQuery 对象。QSqlDatabase 的很多成员函数调用都直接转发给 QSqlDriver 来处理。

QSqlResult 类是定义 SQL 数据库存取功能的抽象基类。QSqlQuery 类提供执行 SQL 语句的手段，包括 SELECT、UPDATE 和 ALTER TABLE 等 DML 语句和 DDL 语句，甚至还可以执行非标准的 SQL 语句。QSqlQuery 的很多成员函数将查询转发给 QSqlResult。

QSqlDriver 和 QSqlResult 类紧密联系，当实现自己的 SQL 驱动时，必须要继承这两个类，并实现所有的抽象虚方法。

为了将 SQL 驱动实现为插件，要在程序中使用 Q_EXPORT_PLUGIN2() 宏导出插件。

Oracle 是使用非常广泛的数据库产品，但在 Qt 开源版中不支持，商业版中也没有完全支持。例如 Qt 商业版的 OCI 驱动对 Oracle 的 Timestamp 数据类型支持得不是很好，不能存储毫秒信息。下面通过简要说明编写 Oracle 数据库驱动插件来说明编写数据库插件的过程，程序中没有使用晦涩难懂的 OCI 接口，而是使用了简洁的 OCCi (Oracle C++ Call Interface) 接口。因为要实现完整的 Oracle 数据库驱动是非常复杂的，所以这里只给出了实现的框架。

为了编写数据库驱动，必须要实现继承两个类：一个是 QSqlDriver，一个是 QSqlResult。Oracle 的 OCCi Driver 类定义如下：

```
class QOCCIDriver : public QSqlDriver
{
public:
    QOCCIDriver(QObject *parent=0) ;
    ~QOCCIDriver();

    bool hasFeature(DriverFeature feature);
    bool open(const QString & db, const QString & user,
              const QString & password, const QString & host,
              int port, const QString & options);
    void close();
    QSqlResult *createResult() const ;
private:
    Environment *env;          /// OCCi 环境
    Connection *conn;          /// OCCi 连接
};
```

为了使用数据库，首先必须要打开数据库，这通过 open() 函数来实现，代码如下：

```
bool QOCCIDriver::open(const QString & db,
                      const QString & user,
                      const QString & password,
                      const QString & host,
```

```

        int port,
        const QString &options)
{
    try {
        conn = env->createConnection(
            (string)user.toLocal8Bit().constData(),
            (string)password.toLocal8Bit().constData(),
            (string)db.toLocal8Bit().constData());
    }
    catch (SQLException &e)
    {
        setLastError(QSqlError(
            QString::fromAscii("Create connection error."),
            QString::fromAscii(e.getMessage().c_str()),
            QSqlError::ConnectionError,
            e.getErrorCode()));
        setOpenError(true);
        return false;
    }
}

```

数据库使用完后，要使用 `close()` 函数关闭。

```

QOCCIDriver::close()
{
    env->terminateConnection(conn);
    Environment::terminateEnvironment(env);
}

```

获取数据库的特性函数 `hasFeature()` 比较简单。

```

QOCCIDriver::hasFeature(DriverFeature feature) const
{
    switch (f) {
        case Transactions:
        case LastInsertId:
        case BLOB:
        case PreparedQueries:
        case NamedPlaceholders:
        case BatchOperations:
        case Unicode:
            return true;
        case QuerySize:
        case PositionalPlaceholders:
            return false;
    }
    return false;
}

```

`QSqlDriver` 中还有其他一些函数需要实现，如 `createResult()`，`tables()`，`commitTransaction()`，



rollbackTransaction()等, 这里不详细展开。

QOCCIResult 类实现数据的操作, 定义如下:

```
class QOCCIResult : public QSqlResult
{
    friend class QOCCIDriver;
public:
    QOCCIResult(const QOCCIDriver * db);
    ~QOCCIResult();
    QSqlRecord rec;

protected:
    QVariant data(int index);
    bool prepare(const QString& query);
    bool exec();
    bool isNull(int index);
    bool reset (const QString& query);
    bool fetch(int index);
    bool fetchFirst();
    bool fetchLast();
    bool fetchNext();
    int size();
    int numRowsAffected();
    QSqlRecord record() const;

private:
    Environment *env;          /// OCCI 环境
    Connection *conn;          /// OCCI 连接
    Statement *stmt;           /// OCCI 语句
    ResultSet* rs;
};
```

由于要实现的函数和要考虑的情况很复杂, 要对 OCCI 编程非常熟悉, 这也不是本书的重点, 这里就不讨论如何实现这些函数。

最后还要实现 OCCI 插件导出类。

```
#include <qsqldriverplugin.h>
#include <qstringlist.h>
#include "qsql_occi.h"

class QOCCIDriverPlugin : public QSqlDriverPlugin
{
public:
    QOCCIDriverPlugin();

    QSqlDriver* create(const QString &);
    QStringList keys() const;
};
```

```

QOCCIDriverPlugin::QOCCIDriverPlugin()
    : QSqlDriverPlugin()
{
}

QSqlDriver* QOCCIDriverPlugin::create(const QString &name)
{
    if (name == QLatin1String("QOCCI") || name == QLatin1String("QOCCIB")) {
        QOCCIDriver* driver = new QOCCIDriver();
        return driver;
    }
    return 0;
}

QStringList QOCCIDriverPlugin::keys() const
{
    QStringList l;
    l.append(QLatin1String("QOCCI"));
    return l;
}

Q_EXPORT_STATIC_PLUGIN(QOCCIDriverPlugin)
Q_EXPORT_PLUGIN2(qsqlocci, QOCCIDriverPlugin)

```

上面的插件类基本是模板，通常需要做的是改变驱动的标志（keys）。

通过上面的流程，可以知道开发数据库插件的过程，要开发真正数据库插件，关键还是要对目标数据库的深入掌握。

18.4 自定义风格插件

在 Qt 中定义了 Mac、Windows XP、Windows Vista、Motif、CDE、Cleanlooks、PlatstiqueStyle 等多种风格。同时还可以通过自定义来实现自己的风格，自定义风格也可以实现为风格插件供其他应用使用。

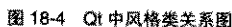
和自定义风格相关的类是 QStyle 和 QStylePlugin，QStyle 类封装了 GUI 外观。Qt 的窗口部件在内部基本上都是用 QStyle 类来完成绘制。QStyleFactory 类创建一个 QStyle 对象。QStyleFactory 类通过 create() 函数和标志风格的一个键来创建风格对象。典型的 key 包括 windows, motif, cde, platstique 和 cleanlooks。windowsxp 和 macintosh 风格只在相应的操作系统上才有效。

Qt 中的几种预定义风格类的继承关系图如图 18-4 所示。

通常实现自定义风格是从 QStyle 子类开始，而不是直接继承 QStyle 类，这样可以利用一些已实现的特性。在实现自定义风格时，QStyle 类中通常可能需要实现的函数有：

1. polish()

polish() 函数有三个重载函数，它在窗口部件创建之后，首次显示之前执行。在不同形式的 polish() 函数中可以改变窗口部件的调色板、窗口部件属性、应用程序的属性。通常还要实现对应的 unpolish() 函数，以便在风格动态切换时恢复原来的设置。



2. drawControl()

drawControl()函数负责绘制特定的元素。例如和 QPushButton 相关的元素有 CE_PushButton、CE_PushButtonBevel 和 CE_PushButtonLabel, 分别是整个按钮、按钮的斜面 and 文本。

3. drawComplexControl()

绘制复杂的窗口部件, 如 CC_SpinBox, CC_ComboBox, CC_ScrollBar 等。

4. drawPrimitive()

绘制原子元素，如 PE_FrameFocusRect（焦点矩形框），PE_IndicatorHeaderArrow（在表格的表头上表示升序或降序的箭头）。

还有其他的一些函数如绘制文本的 `drawItemText()`，绘制图像的 `drawItemPixmap()` 等，可以根据实际选用。这些函数不一定只能在自定义风格类中使用，也可以在 `QWidget` 的 `paintEvent()` 函数中使用，还可以使用 `QStylePainter` 类来绘制。

自定义风格完成后，直接使用 `QApplication::setStyle()` 函数设置就能起作用了。也可以在命令行使用参数来设置，方法如下：

```
./qtapp -style cleanlooks
```

如果将自定义风格做成风格插件，风格就可以供其他应用程序使用了。现在实现一个最简单的风格插件，来学习自定义风格插件的制作。这里的插件是实现文本编辑框变为黑底白字，这种风格可以减轻长时间阅读文本时的眼疲劳。风格的定义如下：

```
#ifndef MYSTYLE_H
#define MYSTYLE_H

#include <QWindowsStyle>
```



```

class MyStyle : public QWindowsStyle
{
    Q_OBJECT

public:
    MyStyle() {};

    void polish(QPalette &palette);
};
#endif

```

MyStyle 直接从 QWindowsStyle 类继承，实现非常简单。

```

#include <QtGui>
#include "mystyle.h"

void MyStyle::polish(QPalette &palette)
{
    palette.setBrush(QPalette::Base, Qt::black);
    palette.setBrush(QPalette::Text, Qt::white);
}

```

polish() 函数改变了调色板，将文本颜色改为白色，背景色改为黑色。如果不实现插件，到这一步就可以使用自定义风格了，这里还要定义插件的接口类。

```

#ifndef MYSTYLEPLUGIN_H
#define MYSTYLEPLUGIN_H

#include <QStylePlugin>

class QStringList;
class QStyle;

class MyStylePlugin : public QStylePlugin
{
    Q_OBJECT

public:
    MyStylePlugin() {};

    QStringList keys() const;
    QStyle *create(const QString &key);
};

#endif

```

在实现中，只需要指定风格的名字为“MyStyle”就可以了。

```

#include <QtGui>
#include "mystyleplugin.h"

```



```
#include "mystyle.h"

QStringList MyStylePlugin::keys() const
{
    return QStringList() << "MyStyle";
}

QStyle *MyStylePlugin::create(const QString &key)
{
    if (key.toLowerCase() == "mystyle")
        return new MyStyle;
    return 0;
}

Q_EXPORT_PLUGIN2(mystyleplugin, MyStylePlugin)
```

最后的 pro 文件中需要说明工程是插件类。

```
TEMPLATE       = lib
CONFIG         += plugin
HEADERS        = mystyle.h \
                mystyleplugin.h
SOURCES        = mystyle.cpp \
                mystyleplugin.cpp
TARGET         = mystyleplugin
DESTDIR        = $$[QT_INSTALL_PLUGINS]/styles

target.path = $$[QT_INSTALL_PLUGINS]/styles
INSTALLS += target
```

现在所有工作已经完成，编译后插件自动将目标文件放在 Qt 的标准风格插件目录中，然后可以编写一个应用程序测试一下。需要注意插件和应用程序的版本必须一致，即都是 Debug 版或 Release 版，否则装入插件会失败。

18.5 小 结

本章学习了 Qt 的插件系统，通过插件可以扩展应用程序和 Qt 自身。可以编写数据库、风格、Qt 设计器等多种形式的插件来扩展 Qt。也能够通过编写应用程序插件来扩展自己的应用程序。对于应用程序插件，本章没有举例，Qt 工具实例中的 plugandpaint 工程和 echoplugin 工程是非常好的应用程序插件实例。

第 19 章 脚本——QtScript

在 Qt 4.3 中，引入了一个新模块——QtScript。应用 QtScript 模块，可以使 Qt 的应用程序可以被脚本操作，同时 Qt 应用程序也可以运行脚本。Qt 支持 ECMA-262 标准定义的 ECMAScript 脚本语言，JavaScript 就是基于 ECMAScript 标准的。所以如果读者熟悉 JavaScript 语言，将有助于理解本章的内容。

19.1 执行 ECMAScript 脚本

要在 Qt 中使用脚本，需要在 qmake 工程文件中加入：

```
QT += script
```

并包含相应的头文件：

```
#include <QtScript>
```

为了学习 QtScript，首先实现一个最简单的 QtScript 程序，能够执行硬编码的 ECMAScript 语句。程序如下：

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());

    QScriptEngine engine;

    QTextEdit textEdit;
    QScriptValue scriptWindow = engine.newQObject(&textEdit);
    engine.globalObject().setProperty("textEdit", scriptWindow);

    QScriptValue text(&engine, QObject::tr(
        "<b><font color=#FF0000 size=5>第一个 QtScript 程序<font></b>"));
    scriptWindow.setProperty("html", text);
    engine.evaluate(QObject::tr("textEdit.append('追加的文字')"));
    engine.evaluate("textEdit.show()");
    if(engine.hasUncaughtException()) {
        QScriptValue value = engine.uncaughtException();
        QMessageBox::information(NULL, QObject::tr("Exception"),
            value.toString());
    }

    return app.exec();
}
```



程序中首先生成了一个 `QScriptEngine` 对象 `engine`。`QScriptEngine` 对象是执行 Qt Script 代码的引擎。`QScriptEngine` 对象的 `evaluate()` 函数可以执行脚本代码，例如：

```
QScriptEngine engine;
QDebug() << "1+2=" << engine.evaluate("1+2").toNumber();
```

上面的代码执行了脚本代码“1+2”，并将结果转换为 Qt 的 `qsreal` 类型。`evaluate()` 函数的返回值是 `QScriptValue`。`QScriptValue` 类包装了 `QScript` 中的各种数据类型，如 `Undefined`、`Null`、`Boolean`、`Number`、`String` 和 `Object` 类型。但 C++ 不能识别这些数据类型，所以 `QScriptValue` 类提供了转换函数 `toInteger()`、`toNumber()`、`toString()`、`toVariant()`、`toQObject()`、`toBoolean()`、`toRegExp()` 等函数将脚本中的类型转换为 C++ 能识别的类型（有些是 Qt 定义的类型）。

接下来使用 `engine.newQObject()` 函数创建了一个 `QScript` 对象包装了 Qt 的 `QTextEdit` 对象。`newQObject()` 函数可以将从 `QObject` 继承的对象创建一个对应的 `QScript` 对象。`QObject` 类型对象的信号、槽、属性以及子对象都可以用相应的 `QScript` 对象存储。

为了使新建的对象在脚本中可用，使用语句：

```
engine.globalObject().setProperty("txtEdit", scriptWindow);
```

将包装的 `QTextEdit` 对象命名为 `txtEdit`。函数 `globalObject()` 返回了脚本的全局对象，它包括了 ECMA-262 定义的 `Math`、`Date` 和 `String` 对象。`setProperty()` 函数完成了定义 `txtEdit` 变量的过程。

为了在 `QTextEdit` 对象中生成文字，可以利用 `QTextEdit` 的 `html` 属性设置文字，`append()` 槽追加文字，然后使用 `show()` 槽显示该窗口部件。

最后进行了异常捕获。如果脚本执行产生错误，将抛出一个异常。`QScriptEngine` 类提供了 `hasUncaughtException()` 函数判断是否有异常抛出，提供了 `uncaughtException()`、`uncaughtExceptionLineNumber()` 和 `uncaughtExceptionBacktrace()` 函数获取异常信息。

第一个 `QScript` 运行结果如图 19-1 所示。

通过这个简单的例子，读者应该了解了 `QScript` 的基本知识，接下来将学习 `QScript` 的详细使用。



图 19-1 第一个 `QScript` 程序

19.2 `QScript` 中的信号和槽

在脚本中可以使用信号和槽，但使用的方式和 C++ 方式不同。在脚本中，信号是一个属性，信号使用 `connect()` 函数和槽函数进行连接。例如：

```
function myFunction()
{
    .....
}
button.clicked.connect(myFunction)
```

如果是和对象的成员函数连接，则使用格式：

```
button.clicked.connect(myobject, myfunction);
```

下面实现一个简单的例子来学习信号和槽的连接，该例子能对 QListWidget 进行清除、增加行或列、删除行或列的功能。

首先使用 Qt 设计器设计 table.ui 文件，如图 19-2 所示。

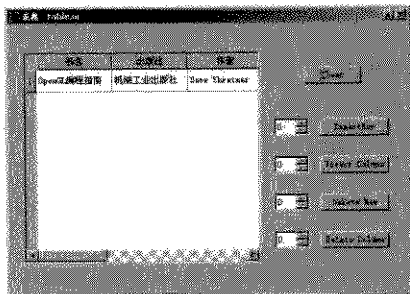


图 19-2 表格 UI 界面

在设计中，顶层窗口部件的对象名为 Table，QTableWidget 的 name 属性为 tableWidget。4 个 spinBox 的 name 属性分别是 spinInsRow、spinInsCol、spinDelRow、spinDelCol。4 个按钮的名称为 btnInsRow、btnInsCol、btnDelRow、btnDelCol。

主程序如下：

```
#include <QtGui>
#include <QtUiTools>
#include <QtScript>

int main(int argc, char **argv)
{
    Q_INIT_RESOURCE(table);

    QApplication app(argc, argv);
    QScriptEngine engine;

    QFile scriptFile(":/table.js");
    scriptFile.open(QIODevice::ReadOnly);
    engine.evaluate(scriptFile.readAll());
    scriptFile.close();

    QUiLoader loader;
    QFile uiFile(":/table.ui");
    uiFile.open(QIODevice::ReadOnly);
    QWidget *ui = loader.load(&uiFile);
    uiFile.close();
```

```

QScriptValue func = engine.evaluate("Table");
QScriptValue scriptUi = engine.newQObject(ui);
QScriptValue table = func.construct(QScriptValueList() << scriptUi);
if(engine.hasUncaughtException()) {
    QScriptValue value = engine.uncaughtException();
    QString lineNumber = QString("\nLine Number:%1\n")
        .arg(engine.uncaughtExceptionLineNumber());
    QStringList btList = engine.uncaughtExceptionBacktrace();
    QString trace;
    for(short i=0; i<btList.size(); ++i)
        trace += btList.at(i);
    QMessageBox::information(NULL, QObject::tr("Exception"),
        value.toString() + lineNumber + trace );
}

ui->show();
return app.exec();
}

```

在主程序中需要注意的一点是使用了 **QtUiTools** 模块, 使用该模块可以在运行时根据 ui 文件动态生成所需的窗口部件对象。为了使用这个功能, 需要在 **qmake** 的 **pro** 文件中加入:

```
CONFIG += uitools
```

并在程序中加入:

```
#include <QtUiTools>
```

生成 ui 对象非常简单, 只需要使用 **QUiLoader** 的 **load()** 函数装入就可以了。

相应的 **ECMAScript** 文件如下:

```

function Table(ui)
{
    this.ui = ui;

    with (ui) {
        btnClear.clicked.connect(this, tableWidget.clear);
        btnInsRow.clicked.connect(this, "insertRow");
        btnDelRow.clicked.connect(this, "deleteRow");
        btnInsCol.clicked.connect(this, "insertCol");
        btnDelCol.clicked.connect(this, "deleteCol");
    }
}

//插入行
Table.prototype.insertRow = function()
{
    with (this.ui) {
        tableWidget.insertRow(spinInsRow.value);
    }
}

```

```

//删除行
Table.prototype.deleteRow = function()
{
    with (this.ui) {
        tableWidget.removeRow(spinDelRow.value);
    }
}
//插入列
Table.prototype.insertCol = function()
{
    with (this.ui) {
        tableWidget.insertColumn(spinInsCol.value);
    }
}
//删除列
Table.prototype.deleteCol = function()
{
    with (this.ui) {
        tableWidget.removeColumn(spinDelCol.value);
    }
}

```

在脚本文件中，将每个按钮的 `clicked()` 信号连接到相应的函数上，在响应函数里调用了 `QTableWidget` 的槽来完成相应的功能。

19.3 使用 JavaScript 操作 Qt 对象

用过 Visual Basic for Application 的用户都知道可以使用 VBA 对 Word、Excel、PowerPoint 等 Microsoft Office 办公软件进行操作和二次编程。同样，有了 QtScript，也可以让 Qt 应用程序实现类似于 VBA 的功能。

为了使 ECMAScript 能够操作对象，需要对 C++ 类进行扩展。所幸的是，Qt 的元对象系统提供了这些手段。从 `QObject` 继承的类或使用了 `Q_OBJECT` 宏的类可以支持运行时的动态特性，如果需要 C++ 类能够支持脚本的操作，只需要简单地从 `QObject` 类继承就可以了。

对于从 `QObject` 继承的类来说，它的信号和槽是可以被脚本操作的。但公共的成员函数脚本不能操作，如果希望一个函数能够被脚本操作，只需要在函数名前加上 `Q_INVOKABLE` 宏就可以了。例如：

```

class scriptObject : public QObject
{
    Q_OBJECT

public:
    Q_INVOKABLE void scriptFunction();
    ...
};

```

有 `Q_INVOKABLE` 修饰的函数也可以在脚本中使用 `connect()` 和信号进行连接。



如果需要将 Qt 的属性显露给脚本程序，要在属性的定义中指明 `SCRIPTABLE` 属性为“true”，例如：

```
Q_PROPERTY(int numCols READ numCols WRITE setNumCols SCRIPTABLE true)
```

但由于 `SCRIPTABLE` 属性默认为“true”，所以也可以不写。

信号和槽的使用，在前面已经讲过。这里只说明一下有参数的信号如何使用，下面的示例代码说明带参数的信号的连接。

```
function copy(bool yes)
{
    print("Copy Available:" + yes);
}

function prepare()
{
    var obj = new MyObject();
    obj["copyAvailable(bool)"].connect(copy);
    ...
}
```

下面通过一个例子来学习如何实现可被脚本操作的 C++ 类。这里直接从 `QTextEdit` 类继承，实现一个 `insertTable()` 函数，能够插入 HTML 表格。该类定义如下：

```
#include <QtGui>
#include <QtScript>

class RTFEdit : public QTextEdit
{
    Q_OBJECT

public:
    RTFEdit(QWidget *parent = 0);
    Q_INVOKABLE void insertTable(int row, int col);
};
```

实现代码如下，实际上就是插入 HTML 代码。

```
#include "rtfedit.h"

RTFEdit::RTFEdit(QWidget *parent)
: QTextEdit(parent)
{
}

void RTFEdit::insertTable(int row, int col)
{
    QString htm("<table border=1 width=100%>");
    for(int i=0; i<row; i++)
    {
        htm += "<tr>";
        for(int j=0; j< col; j++)
        {
```



```

        htm += "<td> </td>";
    }
    htm += "</tr>";
}
htm += "</table>";
insertHtml(htm);
}

```

打开 Qt 设计器，创建一个基于窗口部件的 ui 文件。新建一个 QTextEdit 控件，并将其提升为 RTFEdit。最后以 rtfedit.ui 为文件名保存。

主程序如下：

```

#include <QtGui>
#include <QtUiTools>
#include <QtScript>
#include "rtfedit.h"
// 动态装入并创建用户界面
class RTFUiLoader : public QUiLoader
{
public:
    RTFUiLoader(QObject *parent = 0)
        : QUiLoader(parent)
    { }

    virtual QWidget *createWidget(const QString &className,
        QWidget *parent = 0, const QString &name = QString())
    {
        if (className == QLatin1String("RTFEdit")) {
            QWidget *rtf = new RTFEdit(parent);
            rtf->setObjectName(name);
            return rtf;
        }
        return QUiLoader::createWidget(className, parent, name);
    }
};

int main(int argc, char **argv)
{
    Q_INIT_RESOURCE(rtfedit);

    QApplication app(argc, argv);
    QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
    QScriptEngine engine;

    QFile scriptFile(":/rtfedit.js");
    scriptFile.open(QIODevice::ReadOnly);
    engine.evaluate(QObject::tr(scriptFile.readAll()));
    scriptFile.close();

    RTFUiLoader loader;

```



```

QFile uiFile(":/rtfedit.ui");
uiFile.open(QIODevice::ReadOnly);
QWidget *ui = loader.load(&uiFile);
uiFile.close();

QScriptValue func = engine.evaluate("RTF");
QScriptValue scriptUi = engine.newQObject(ui);
QScriptValue table = func.construct(QScriptValueList() << scriptUi);
if(engine.hasUncaughtException()) {
    QScriptValue value = engine.uncaughtException();
    QString lineNumber = QString("\nLine Number:%1\n")
        .arg(engine.uncaughtExceptionLineNumber());
    QStringList btList = engine.uncaughtExceptionBacktrace();
    QString trace;
    for(short i=0; i<btList.size(); ++i)
        trace += btList.at(i);
    QMessageBox::information(NULL, QObject::tr("Exception"),
        value.toString() + lineNumber + trace );
}

ui->show();
return app.exec();
}

```

程序中使用了动态加载 UI 的类，由于 RTFEdit 是自定义的类，所以需要实现自己的 UiLoader 类。在 RTFUiLoader 中，创建了自己的 RTFEdit 对象。

JavaScript 程序很简单，主要是测试程序是否有效。

```

function RTF(ui)
{
    this.ui = ui;
    with (ui) {
        rtfEdit.insertHtml('<h1 align=center>标题</h1>');
        rtfEdit.insertTable(4, 4);
    }
}

```

程序运行效果如图 19-3 所示。

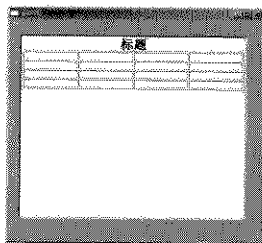


图 19-3 可被脚本操作的 C++ 对象

19.4 基于 Prototype 的继承

JavaScript 虽然也是面向对象的语言，但是和 C++ 以类为基础不同的是，JavaScript 是基于原型 (prototype) 的。原型本身也是一个对象，如果说几个对象共享同一个原型，那么它们是同一类的对象。当访问对象的属性时，如果对象自身没有这个属性，那么它将在它的原型中查找，如果原型中没有，则在原型的原型中查找，直到找到该属性或原型为 null 为止 (Object 的原型为 null)。

在 JavaScript 中，定义类是隐式的，即定义一个函数就是定义了一个类。例如：

```
function Book(name, price)
{
    this.name = name;
    this.price = price;
}
```

上面的代码就定义了类 Book，它的原型是 Book.prototype。因为 JavaScript 是动态语言，接下来可以给 prototype 定义新的函数。

```
Book.prototype.discount = function(dis)
{
    this.price = this.price * dis;
}
```

在 QtScript 中，可以用 QScriptEngine::setDefaultPrototype() 函数给 C++ 类指定一个自定义的原型，这样脚本就可以通过原型类访问 C++ 的类了。还可以使用 QScriptEngine::newFunction() 函数将 C++ 的函数包装给脚本使用。

这里只是简单地介绍了一下原型，要掌握原型的使用，可以参考 Qt 脚本示例中的 defaultprototypes 工程。

19.5 小 结

QtScript 是 Qt 4.3 中的新功能模块。通过 QtScript，应用程序可以被脚本操作，能够实现一些比较灵活的功能。可以将 C++ 对象的成员函数、信号等显露给 ECMAScript，实现 C++ 与 ECMAScript 的互操作。对于 ECMAScript 中的原型，Qt 也支持设置 C++ 类的原型。

第 20 章 国际化

国际化就是使应用程序能够支持不同的国家和语言。Qt 内置的国际化支持使得程序员只做简单的工作就可以完成国际化过程。

Qt 支持大多数现代语言，如：

- 东亚语言（汉语、日语和韩语）；
- 西方语言；
- 阿拉伯语；
- 斯拉夫语（俄语、乌克兰语）；
- 希腊语；
- 希伯来语；
- 泰语和老挝语；
- Unicode 4.0 中需要特殊处理的语言。

在 Windows NT/2000/XP 和支持 FontConfig（客户端字体支持）的 UNIX/X11 系统及 Qtopia Core 中还支持其他的一些语言。

Qt 提供了 Qt Linguist 工具来帮助对应用程序进行翻译的管理，简化了国际化的工作量和过程。

20.1 Unicode 与字符编码

在处理字符串、文本文件时，不可避免地要遇到各种编码，如 UTF-8、GBK、ISO 10646、ISO 8859 等。各种编码让人头疼不堪，经常出现汉字不能正常显示等令人烦恼的情况。这里，简略复习一下和汉字有关的编码，并学习在 Qt 中如何处理各种编码。

20.1.1 Unicode

Unicode 是统一表示各种语言字符的编码标准，它能够跨平台、跨程序、跨语言。Unicode 最新的标准是 5.0 版，Qt 4 采用的 Unicode 4.0 标准。Unicode 编码只与 ISO-8859-1 兼容，和汉字编码 GB 系列标准不兼容。

Unicode 只是分配了字符编码，UTF（UCS Transformation Format）规范规定了 Unicode 编码的具体表示方式，常见的 UTF 规范包括 UTF-8、UTF-16 等。

UTF-8 就是以 8 位为单元对 UCS 进行编码，UTF-8 可以兼容 ASCII 字符。UTF-16 以 16 位为单元对 UCS 进行编码。对于小于 0x10000 的 Unicode 编码，UTF-16 编码就等于 Unicode 编码对应的 16 位无符号整数。UTF-16 和 ASCII 字符不兼容。

UTF-8 以字节为编码单元，没有字节序的问题。UTF-16 以两个字节为编码单元，在解释一个 UTF-16 文本前，首先要弄清楚每个编码单元的字节序。Unicode 规范中推荐的标记字节顺序的方法是 BOM（Byte Order Mark）。即在文本文件的最开始用 FEFF 表示文件是 Big-Endian 的；如果是 FFFE，

就表明该文件是 Little-Endian 的。UTF-8 不需要 BOM 来表明字节顺序,但可以用 BOM 来表明编码方式。在文件开始使用 EF BB BF 表示 UTF-8 编码。

在 Qt 中, QChar 类可以表示 Unicode 字符,并提供了各种处理函数。同样, QString 也是支持 Unicode 的字符串类。如果只表示 ASCII/Latin-1 字符或字符串,可以使用 QLatin1Char 和 QLatin1String 类。 QTextStream 类能够对字符流进行自动检测并识别 Unicode 编码类型。

20.1.2 汉字编码

常见的汉字编码也有好几种,如 GB2312、GBK、GB18030、BIG5 等。这里简要地说明一下简体中文常见的几种编码。

GB2312-80 一共收录了 7 445 个字符,包括 6 763 个汉字和 682 个其他符号。

1995 年的汉字扩展规范 GBK1.0 收录了 21 886 个符号,它分为汉字区和图形符号区。汉字区包括 21 003 个字符。

GB18030-2000 是取代 GBK1.0 的正式国家标准。该标准收录了 27 484 个汉字,同时还收录了藏文、蒙文、维吾尔文等主要的少数民族文字。现在的 PC 平台必须支持 GB18030,对嵌入式产品暂不作要求。

从 ASCII、GB2312、GBK 到 GB18030,这些编码方法是向下兼容的,即同一个字符在这些方案中总是有相同的编码,后面的标准支持更多的字符。在这些编码中,英文和中文可以统一处理。

20.1.3 编码转换

由于 Qt 支持多种编码,所以编码转换工作也非常轻松。通常在 Windows 下 Eclipse 的默认编码是 GB18030 或 GBK,而到了 Linux 下,默认的编码可能又是 UTF-8。这样,有时在 Windows 下编写好的程序到了 Linux 下汉字无法正确识别。下面实现一个利用 Qt 的编码功能将 GB18030 和 UTF-8 的源程序进行互相转换程序。

程序的界面如图 20-1 所示,界面是使用 Qt 设计器设计的,文件名为 convert.ui。

在 ui 文件中,源路径和目标路径的检查框分别命名为 txtSource、txtDest。两个浏览按钮的对象名是 btnSource、btnDest。依据 ui 文件,窗口定义如下:

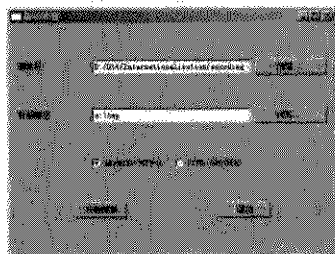


图 20-1 编码转换程序界面

```
#ifndef CONVERTWIDGET_H_
#define CONVERTWIDGET_H_

#include <QWidget>
```

```

#include "ui_convert.h"

class ConvertWidget : public QWidget, private Ui::convertWidget
{
    Q_OBJECT

public:
    ConvertWidget(QWidget *parent = 0);
    virtual ~ConvertWidget();
private slots:
    void on_btnConvert_clicked();
    void on_btnExit_clicked();
    void on_btnSource_clicked();
    void on_btnDest_clicked();
    void on_radioGBToUTF_clicked();
    void on_radioUTFToGB_clicked();
private:
    QTextCodec *srcCodec;
    QTextCodec *destCodec;
};

#endif /*CONVERTWIDGET_H*/

```

在定义中使用了自动连接信号和槽的方式，即定义如下的槽：

```
void on_<widget name>_<signal name>(<signal parameters>);
```

在构造函数中对变量进行初始化。

```

ConvertWidget::ConvertWidget(QWidget *parent)
: QWidget(parent)
{
    setupUi(this);
    txtSource->setText(QDir::currentPath());
    srcCodec = QTextCodec::codecForName("GB18030-0");
    destCodec = QTextCodec::codecForName("UTF-8");
}

```

当单击“开始转换”按钮时，执行转换函数。

```

void ConvertWidget::on_btnConvert_clicked()
{
    QFile srcFile, dstFile;
    QTextStream in, out;
    QString fileName, dstFileName, tmpStr;
    QDirIterator it(txtSource->text(), QDirIterator::Subdirectories);
    while (it.hasNext()) {
        fileName = it.next();
        // QDirIterator 总是返回分隔符 "/"
        fileName.replace("/", QDir::separator());
        if(fileName.endsWith(".h") || fileName.endsWith(".cpp"))
        {
            srcFile.setFileName(fileName);

```

```

        if (!srcFile.open(QFile::ReadOnly | QFile::Text)) {
            QMessageBox::warning(this, tr("错误"),
                tr("不能打开文件%1:\n%2")
                    .arg(fileName)
                    .arg(srcFile.errorString()));
            return;
        }

        in.setDevice(&srcFile);
        in.setAutoDetectUnicode(false);
        in.setCodec(srcCodec);
        tmpStr = in.readAll();

        dstFileName = txtDest->text() + QDir::separator() +
            fileName.mid(fileName.lastIndexOf(QDir::separator())+1);
        dstFileName.replace("/", QDir::separator());
        dstFile.setFileName(dstFileName);
        if (!dstFile.open(QFile::WriteOnly | QFileDevice::Truncate
            | QFile::Text)) {
            QMessageBox::warning(this, tr("错误"),
                tr("不能打开文件%1:\n%2")
                    .arg(dstFileName)
                    .arg(dstFile.errorString()));
            return;
        }

        out.setDevice(&dstFile);
        out.setCodec(destCodec);
        out << tmpStr;
        srcFile.close();
        dstFile.close();
    }

    QMessageBox::information(this, tr("提示"), tr("转换完毕!"));
}

```

函数中使用 `QTextStream` 处理文件流，并关闭了 `QTextStream` 自动检测 Unicode 的特性。在读入流时，将流编码设为一种格式，在写入流时，设为另外一种格式，这样就完成了编码转换。

20.2 Qt Linguist

在一些情况下，国际化是相当简单的。如对移植美国英语到英国英语的应用程序，只需要修改少量的单词。但如果要将英语应用移植为中文应用，就相对复杂一些，这涉及语言的不同、输入法的不同、字符编码的不同和显示方式的不同。

所幸的是 Qt 在国际化方面做了很多工作，因此应用开发者的负担较小。输入法和不同语言的文字显示在 Qt 中是自动处理的。Qt 的字体引擎能够处理不同的语言混合使用。同时 Qt Linguist (Qt 语言学家) 工具提供了支持多种语言翻译的功能。使用 Qt Linguist 翻译过程如下：

01 在 Qt 的用户界面等处使用的本地语言文本都存储在源代码中。Qt 为每个使用的短语都提供了上下文信息，程序员负责在需要的地方加入上下文信息（使用 `tr()`）：

02 发布管理器从源代码产生翻译文件，并传递给翻译人员；

03 翻译人员使用 Qt Linguist 打开翻译文件，输入翻译文本并存入翻译文件，该文件再传回发布管理器；

04 发布管理器再生成二进制的翻译文件给具体应用使用。

随着应用开发不断演化并改变，翻译过程也要相应地重复这些流程。为了保证短语的一致翻译，Qt Linguist 还提供了一个短语参考工具在多个应用和工程之间保证一致的翻译。

不同的语言都存在着许多差异，例如：

- 一个短语可能在目标语言的不同上下文中有不同的表示；
- 键盘快捷键在不同的语言中可能不相同；
- 短语中包含的阿拉伯数字在不同的语言中位置可能不一致。

Qt Linguist 考虑到了这些差异，并提供了相应的解决办法。

下面结合实例，通过将第 6 章中的绘图程序翻译为中文的应用程序来熟悉使用 Qt Linguist 的步骤和方法。

20.2.1 发布管理器

发布管理器主要完成发布应用的功能。应用程序在开发完毕和翻译完毕后需要进行发布。发布管理器包括两个工具，`lupdate` 和 `lrelease`。`lupdate` 工具用来同步源代码和翻译，`lrelease` 工具用来创建发布应用使用的运行时翻译文件。这两个工具依赖于 `qmake` 工程文件。

1. 在 `qmake` 文件中加入指令

为了使用 `lupdate` 和 `lrelease`，在 `qmake` 文件中必须有 `TRANSLATIONS` 关键字指明每种要翻译的语言文件。将第 6 章中的 `basicdraw` 工程改名为 `basicdrawI18N`，并在 `.pro` 文件中加入：

```
TRANSLATIONS = basicdraw_zh_CN.ts
```

通常在语言文件后面加上区域代码，以便区分。中文的区域代码是 `zh_CN`。

在程序中，通常使用 `QTextCodec::setCodecForTr()` 使得 `tr()` 函数中的字符串可以使用 8 位的编码。例如在源语言是中文的情况下，如果没有设置编码，`tr()` 使用 `Latin1`。

如果在应用程序中没有使用 `QTextCodec::codecForTr()` 机制，则需要在 `.pro` 文件中设置 `CODECFORTR` 选项，例如：

```
CODECFORTR=GB18030
```

如果编译器的源代码和运行时系统的编码不同，而且在字符串中要使用非 ASCII 字符，则需要使用 `CODECFORSRC`。如下：

```
CODECFORSRC=UTF-8
```

只有 Microsoft Visual Studio 2005 .NET 需要进行这样的设置。

2. 使用 `lupdate` 生成翻译文件

程序编写完毕后，需要使用 `lupdate` 工具生成翻译文件。`lupdate` 读取 `.pro` 文件，从指定的源文件和

Qt 设计器界面文件中寻找需要翻译的字符串, 并产生或更新工程文件中的 .ts 文件。产生的 .ts 文件可以由 Qt Linguist 打开并填入相应的翻译。

lupdate 可以随着应用程序的开发并行使用, 这样可以把翻译任务分解到开发过程中, 而不用在最后集中翻译。

.ts 文件的是用 XML 表示的, 这样配置管理工具就可以和源程序一样地对待 .ts 文件, 可以进行比较、增量存储等操作。

如果改变了工程 basicdrawI18N, 则需要运行 lupdate。

```
lupdate basicdrawI18N.pro
```

输出结果可能如下所示:

```
Updating 'basicdraw_zh_CN.ts'...
Found 51 source texts (1 new and 50 already existing)
```

结果说明此次更新中有一条新的翻译, 原来有 50 条已经生成的翻译。

3. 用 lrelease 生成二进制文件

lrelease 工具读取 .pro 工程文件并产生 .qm 文件, 其中每一个 .ts 文件产生一个 .qm 文件。qm 文件格式是二进制的, 查找的速度非常快, 在程序运行时使用。

该工具在应用程序每次发布时运行, 运行方法如下:

```
lrelease basicdrawI18N.pro
```

输出结果可能如下:

```
Updating 'E:/Qt4/Internationalization/basicdrawI18N/basicdraw_zh_CN.qm'...
Generated 39 translations (23 finished and 16 unfinished)
Ignored 12 untranslated source texts
```

输出结果说明 lrelease 产生了 39 条翻译, 其中有 23 条已完成, 16 条未完成。忽略了 12 条未翻译的条目。最后中文界面的绘图程序如图 20-2 所示。

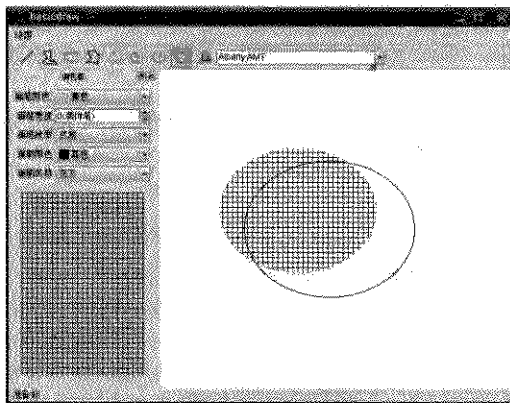


图 20-2 中文界面的绘图程序



20.2.2 翻译器

翻译人员在翻译条目时要使用到 Qt Linguist 工具，Qt Linguist 翻译器界面如图 20-3 所示。

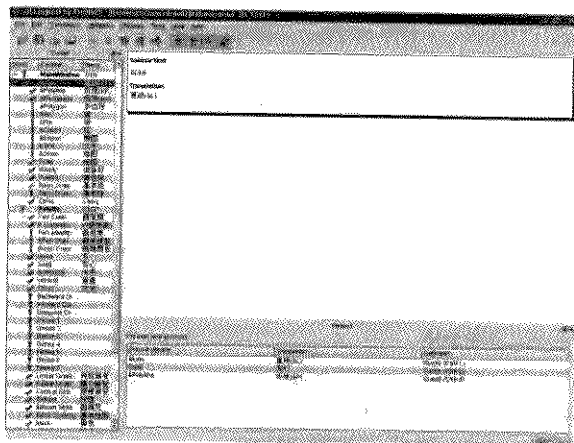


图 20-3 Qt Linguist

Qt Linguist 的执行文件名为 `linguist`，可以在命令行中直接键入“`linguist`”即可运行。要编辑一个翻译文件打开相应的 `.ts` 源文件即可。

如图 20-3 所示，Qt Linguist 的主窗口分为 4 个区域。左边是上下文列表，右上区域是源文本和翻译文本区域，右下部分是短语和猜测（guess）区域。Qt Linguist 是用来编辑翻译文本的工具。上下文在 Qt Linguist 中是指在屏幕上同时出现的一组短语，如在同一个菜单或对话框中的短语。

为了编辑一个上下文，在左侧区域单击上下文，然后在右侧区域出现源文字，这时就可以在翻译区域输入翻译文本了。翻译完成后按下 `Ctrl+Enter` 键提交结果并移到下一个需要翻译的短语。

Linguist 在短语和猜测区域提供了在 Linguist 翻译文件或短语手册中近似的翻译。每一个猜测的翻译都有快捷键，以 `Ctrl+1`、`Ctrl+2` 的形式指定快捷键。可以使用鼠标或快捷键选择已有的翻译直接使用或进行修改。

翻译完成后，选择“File”菜单下的“Save”或使用工具栏上的“保存”按钮进行保存。

Qt Linguist 的主窗口分为 4 个区域。

1. 上下文列表

上下文列表是一个树控件。树控件的顶层条目对应着源文件中的一个类，第二层条目对应着相应的翻译条目。树控件的第一列“Done”，说明翻译是否完成。对号表示所有的翻译已经完成并且是正确的；黄色的问号表示有部分翻译没有完成或没有验证正确性；青色的问号表示尚未翻译；感叹号表示翻译未完成；灰色的对号表示过时的翻译，即以以前的版本中存在的翻译，但在新版本中没有。第二列“Context”是翻译短语出现的上下文。第三列“Items”对不同级别的条目显示不同内容。对树的顶层条目，该列显示了两个数字，第一个数字是有多少个翻译条目已完成，第二个数字是指该上下文中有多少个短语。如果两个数字相等则表示该上下文中所有的短语都已经翻译完成。树的第二层条目中，

该列显示相应的翻译文字。

上下文是按字母排序的。上下文中的短语则是按照它们在源程序中出现的顺序排列的。

上下文列表是可停靠窗口，它可以拖出主窗口或停在主窗口其他位置，而且 Qt Linguist 可以记住上下文窗口的位置并在下次启动时恢复上次的位置。

2. 源文字区域

Qt Linguist 提供了 4 种类型的验证，快捷键、结尾标点、短语和占位符。如果源文本包括快捷键，如符号“&”，但目标翻译文本不包含“&”，则快捷键验证失败。同样的，如果源文本以标点符号结束，如“?”、“!”、“.”，但翻译文本不以这些符号结尾，则验证失败。如果源文本的短语在短语参考(open book)中，但翻译文本和短语参考翻译不一致，则短语验证失败。

例如在翻译“Pen & Width:”条目过程中，如果快捷键没有对应翻译(即“&W”)，在状态栏中显示“Accelerator possibly missing in translation”，告知没有翻译快捷键。如果忘了条目最后的冒号，状态栏提示“Translation does not end with the same punctuation as the source text”，告知翻译文本和源文本结尾没有相同的标点符号。

如果不想验证其中的某项，可以关闭。选择菜单“Validation”中的菜单项就可以关闭相应的验证规则。

3. 翻译区域

翻译区默认在主窗口的右上部。由三个垂直的部分组成。最上面的是“Source Text”，显示源文本。中间是程序员用来提示翻译的高亮蓝色背景文本。如果没有上下文信息，则本栏不出现。第三栏是输入翻译文本的地方。

4. 短语和猜测区

该区域默认在主窗口右下方。如果在短语参考中有当前短语或在文件中有类似的短语，则出现在该区域。这些猜测有时并不是很准确，甚至风牛马不相及。

在使用 Qt Linguist 过程中，如果想推迟一个翻译可以按 Ctrl+L (下一个未完成)，然后移到下一个未完成的翻译。未完成的翻译包括没有翻译的或验证失败的。移到下一个短语是 Shift+Ctrl+L。当然这些操作也可以使用工具栏和菜单来完成。

在多个上下文中，短语的翻译可以是一致的。当一个短语翻译后，下次再次出现该短语时，Qt Linguist 将会提供以前的翻译作为候选翻译在短语和猜测区。如果之前的翻译是可以接受的，只需要单击“Done & Next”按钮(或按 Ctrl+Return)就可以了。

如果一个短语在一个上下文中出现多次，Qt Linguist 的上下文列表中 will 只显示一次，翻译也是相同的。如果该短语在相同的上下文中有不同的翻译，程序员则必须为其提供不同的注释。程序员的注释将以淡蓝色显示。注释写在 QObject::tr() 函数中，如：

```
setWindowTitle(tr("Basic Draw Demo", "Window title"));
```

QObject::tr() 函数的第二个参数就是注释，可以给翻译人员一些提示。

在使用 Qt Linguist 时，还可以完成如下一些任务。

1. 改变键盘快捷键

键盘快捷键通常是由 Alt 或 Ctrl 键与其他键组合而成，在菜单和按钮上使用。如果一个字母下面

有下划线, 则代表同时按下 Alt 键和该字母键等同于鼠标单击该按钮或菜单。Ctrl 键组成的快捷键通常独立于控件, 它既可以是菜单上的一项, 也可以不和菜单关联。

在翻译时, 这些快捷键可以根据目标语言的需要进行改变。

2. 处理包含变量的短语

一些短语中包含运行时才确定的变量, 如下文本所示。

当处理完文件%1后, %2将要打开。

在这里, “%1”是第一个参数, “%2”是第二个参数。在翻译文本中, 这些参数仍然存在, 但位置可能有变化。在翻译时可以改变“%1”和“%2”的位置, 但它们代表的意义不变。

3. 复用翻译

如果翻译文字和源文字一致, 可以使用“Translation”菜单中的“Begin from Source”菜单(或 Ctrl+B)将源文字拷到翻译文字区。

4. 创建和使用短语参考

Qt Linguist 短语参考实际上是一些源语言短语、翻译的短语和可选的定义组成的短语翻译集合。短语参考和具体应用程序无关。

如果未翻译的短语和短语参考中的相同, Qt 将会在相关短语面板显示短语手册中相应的条目。如果翻译和短语手册中不一致, 将显示一个问号。短语手册可以用来保证翻译的一致性。

创建一个短语手册的方式是选择“Phrase→New Phrase Book”菜单, 然后输入文件名(.qph 为扩展名)。打开已有的短语手册则是选择“Phrase→Open Phrase Book”。

添加一条新短语为单击对话框中的“New Phrase”按钮, 输入源文本和翻译文本以及可选的定义(用来区分同一条短语的多种翻译)。

20.2.3 加载翻译文件

在 Qt 应用程序中支持多语言是很简单的, 程序员只需在窗口或对话框创建时装入翻译的字符串就可以了。

在应用程序开发过程中, 翻译的迭代过程通常如下:

- 01 运行 lupdate 工具生成工程的翻译源文件(.ts);
- 02 用 Qt Linguist 打开源文件并输入翻译, Qt 对文本进行管理;
- 03 运行 lupdate 更新加入的新文本, 同步源代码和翻译文件;
- 04 重复步骤 2 和步骤 3;
- 05 使用 lrelease 发布翻译文件, 将.ts 文件生成相应的.qm 文件。

应用程序员可以通过两种方式加载翻译文件, 一种是硬编码方式, 直接指定加载的语言, 代码如下:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTranslator translator;
    translator.load("basicdraw_zh_CN");
    app.installTranslator(&translator);
}
```

另外一种自动判断翻译当前的 locale, 再装入相应的翻译文件, 如下所示:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QString locale = QLocale::system().name();
    QTranslator translator;
    translator.load(QString("basicdraw_") + locale);
    app.installTranslator(&translator);
}
```

20.3 语言切换

通常情况下，一个应用程序的界面语言在启动时就已经确定了。但在 Qt 的应用程序中可以实现语言运行时动态切换，满足一些特定的要求。

实现动态语言切换的关键是将所有需要切换的字符串放在一个函数中，如 `retranslateUi()`。如果看过 Qt 的 `uic` 生成的头文件，就知道 Qt 的 `uic` 就是这样处理的。在需要切换语言时，加载相应的语言文件，并调用 `retranslateUi()` 函数就可以了。

另外也可以通过监听 `LocaleChange` 和 `LanguageChange` 事件来感知语言的变化，并在事件处理函数中重新设置用户界面上的文字。例如：

```
void MyWidget::changeEvent(QEvent *event)
{
    if (e->type() == QEvent::LanguageChange) {
        myLabel->setText(tr("My Label"));
        ...
        myPushButton->setText(tr("&PushButton"));
    } else
        QWidget::changeEvent(event);
}
```

这种方式需要事先装入相应语言的翻译文件。

20.4 小 结

Qt 支持多种语言和编码，支持 UTF-8 和 UTF-16，提供了处理多种编码的类和函数，这为处理多语言提供了极大的方便。Qt Linguist 工具为多语言应用的翻译提供了强大的功能支持，通过 Qt Linguist 可以方便地管理翻译文本。还可以通过监听语言变化事件，实现动态语言切换。

第 21 章 Qt 单元测试框架

JUnit 是由 Erich Gamma 和 Kent Beck 编写的一个 Java 回归测试框架 (regression testing framework)。JUnit 测试是由程序员完成的单元测试, 属于白盒测试, 因为程序员知道被测试的软件如何完成功能和完成什么样的功能。程序员通过继承 TestCase 类, 就可以用 JUnit 进行自动测试了。

JUnit 推出之后, 风靡了整个单元测试领域, 于是诞生了 CppUnit, NUnit, HttpUnit, DUnit, PHPUnit 等适合特定语言的各种 XUnit 测试框架。Qt 的单元测试 QTestLib 也类似于这些单元测试工具。

21.1 QTestLib 框架

21.1.1 QTestLib

QTestLib 是用来对基于 Qt 的应用程序和库进行单元测试的框架。QTestLib 既可以进行数据驱动测试, 也可以进行 GUI 测试, 并可以集成到 IDE 中。QTestLib 还可以完成控制台程序测试。QTestLib 的所有的公共方法和类型都在 QTest 名字空间。

创建一个测试的步骤是, 继承 QObject 类并添加私有的槽。每一个私有的槽是一个测试函数。然后使用 QTest::qExec() 来执行测试对象中的所有测试函数。

在测试类中, 有 4 个私有槽有特定的用处, 它们不是测试函数, 如下:

- initTestCase(): 在第一个测试函数运行前调用;
- cleanupTestCase(): 在最后一个测试函数运行完后调用;
- init(): 在每个测试函数运行前调用;
- cleanup(): 在每个测试函数运行后调用。

如果 initTestCase() 运行失败, 将没有测试函数运行。如果 init() 失败, 紧随其后的测试函数将不会被执行, 将直接运行下一个测试函数。

为了使用 QTestLib, 必须在 qmake 中的 .pro 文件中加入:

```
CONFIG += qtestlib
```

在源文件中加入:

```
#include <QTest>
```

21.1.2 第一个 Qt 单元测试

为了展示 Qt 的单元测试功能, 首先通过一个最简单的例子来说明如何使用单元测试。在这里实现了一个能够将海里制转换为公里制的类, 然后编写测试代码来检查该类能否完成相应的功能。被测试类如下:

```
#include <QObject>
```

```

class Nautical : public QObject
{
    Q_OBJECT

public:
    Nautical(int nm)
    {
        m_nm = nm;
    };

    double toKilometer()
    {
        return m_nm * 1.852;
    }

private:
    double m_nm;
};

```

在 `Nautical` 类中，函数 `toKilometer()` 完成将海里制转换为公里制的功能。为了测试转换函数是否正确，编写了如下的测试代码。

```

#include <QtTest/QtTest>
#include "nautical.h"

using namespace QTest;

class TestNautical: public QObject
{
    Q_OBJECT
private slots:
    void toKilometerTest();
};

void TestNautical::toKilometerTest()
{
    Nautical nautical(1);
    QVERIFY(qAbs(nautical.toKilometer() - 1.852) < 0.0000001);
}

QTEST_MAIN(TestNautical)
#include "testnautical.moc"

```

在测试类中 `TestNautical` 中，私有槽 `toKilometerTest()` 就是测试函数。在函数中，初始化对象为 1 海里，然后使用了 `QVERIFY()` 宏来判断 1 海里是否被转换为 1.852 公里。由于浮点数不能直接比较，所以取转换值和实际值的绝对值，只要两者之差小于 10^{-7} ，就认为结果是正确的。

`QVERIFY()` 宏检查表达式是否为真，如果表达式为真，则程序继续运行，否则测试失败，程序运



行终止。如果需要在失败的时候输出信息，可以使用 QVERIFY2()宏，用法如下：

```
QVERIFY2(condition, message);
```

QVERIFY2()在“condition”条件验证失败时，输出信息“message”。

对于可以直接比较的类型，则可以使用 QCOMPARE(actual, expected)宏。QCOMPARE 使用“等号”操作符比较实际值(actual)和期望值(expected)。如果两个值相等，程序继续执行。如果两个值不相等，产生一个错误，而且程序不再继续执行。

QTEST_MAIN()宏实现 main()函数，并初始化 QApplication 对象和测试类，然后按照测试函数的运行顺序执行所有的测试。

测试函数运行结果如下：

```
***** Start testing of TestNautical *****
Config: Using QTest library 4.3.2, Qt 4.3.2
PASS : TestNautical::initTestCase()
PASS : TestNautical::toKilometerTest()
PASS : TestNautical::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped
***** Finished testing of TestNautical *****
```

21.2 数据驱动测试

在上节的例子中，完成了第一个单元测试，并测试了一项数据。但在实际测试中，要对多种边界数据进行测试，需要逐项初始化，逐项完成测试，但这样非常烦琐。QTestLib 考虑到了这一点，可以使用 QTest::addColumn()函数建立要测试的数据列，QTest::newRow()函数添加数据列。初始化数据的函数名和测试函数名一样，但增加了后缀_data()。这里通过一个例子来说明如何进行批量数据测试，被测试类还是上一节的 Nautical 类，但需要在类定义之后加上如下代码。

```
Q_DECLARE_METATYPE(Nautical)
```

该宏将 Nautical 定义为元类型，这样所有基于模板的函数可以使用 Nautical，而 QTest 中用到了模板函数 addColumn()，因此必须使用 Q_DECLARE_METATYPE()宏让模板函数可以识别 Nautical 类。

测试代码如下：

```
#include <QTest/QTest>
#include "nautical.h"

using namespace QTest;

class TestNautical: public QObject
{
    Q_OBJECT

private slots:
    void toKm_data();
    void toKm();
};
```



```

void TestNautical::toKm_data()
{
    // 定义测试数据列
    QTest::addColumn<Nautical>{"nautical"};
    QTest::addColumn<double>{"kilometer"};

    // 建立测试数据
    QTest::newRow("1") << Nautical(1) << 1.852;
    QTest::newRow("10") << Nautical(10) << 18.52;
    QTest::newRow("100") << Nautical(100) << 185.2;
}

void TestNautical::toKm()
{
    // 取测试数据
    QFETCH(Nautical, nautical);
    QFETCH(double, kilometer);

    QVERIFY(qAbs(nautical.toKilometer() - kilometer < 0.0000001));
}

QTEST_MAIN(TestNautical)
#include "testdata.moc"

```

这里，建立了两列数据，`nautical` 列为 `Nautical` 对象，`kilometer` 列是相应的 `Nautical` 对象中海里数转换为公里数据的期望值。测试数据通过 `QTest::newRow()` 函数加入。

在测试函数 `toKm()` 中，通过 `QFETCH()` 宏获取所有数据。然后 `QVERIFY()` 宏将会根据数据有多少行运行多少次。

测试运行输出如下：

```

***** Start testing of TestNautical *****
Config: Using QTest library 4.3.2, Qt 4.3.2
PASS : TestNautical::initTestCase()
PASS : TestNautical::toKm()
PASS : TestNautical::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped
***** Finished testing of TestNautical *****

```

21.3 GUI 测试

21.2.1 仿真 GUI 事件

`QTestLib` 提供了测试 GUI 程序的能力。测试中 `QTestLib` 发出 Qt 事件而不是底层事件（如 X11 事件或 Windows 事件），这样测试程序不会对其他应用程序产生干扰。例如产生键盘事件的函数如下：

```

void QTest::keyClicks ( QWidget * widget, const QString & sequence, Qt::KeyboardModifiers
modifier = Qt::NoModifier, int delay = -1 )

```



keyClicks()函数在指定的窗口部件上产生一系列的键盘事件，并可以使用修饰键（Ctrl, Alt, Shift等），指定延迟时间。

通过测试 Qt 中的 QSpinBox 对象来说明如何进行 GUI 测试。在例子中，通过模拟键盘上的向上和向下光标键来改变 QSpinBox 中的值，测试 QSpinBox 是否能正确改变相应的数值。

测试代码如下：

```
#include <QtGui>
#include <QtTest/QtTest>

class TestGui: public QObject
{
    Q_OBJECT

private slots:
    void testGui();
};

void TestGui::testGui()
{
    QSpinBox spinBox;

    spinBox.setMinimum(1);
    spinBox.setMaximum(100);
    spinBox.setSingleStep(1);
    spinBox.setValue(1);

    QTest::keyClick(&spinBox, Qt::Key_Up);
    QCOMPARE(spinBox.value(), 2);
}

QTEST_MAIN(TestGui)
#include "guitest.moc"
```

在测试函数 testGui()中模拟了 Qt::Key_Up 事件。

QTest::keyClick(&spinBox, Qt::Key_Up)

这相当于在 spinBox 对象上按下了键盘上的向上光标键。由于 spinBox 的初始值为“1”，所以现在应该为“2”，这里使用了 QCOMPARE()宏来比较实际值和期望值是否相等。

测试输出如下：

```
***** Start testing of TestGui *****
Config: Using QTest library 4.3.2, Qt 4.3.2
PASS : TestGui::initTestCase()
PASS : TestGui::testGui()
PASS : TestGui::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped
***** Finished testing of TestGui *****
```

21.2.2 重放 GUI 事件

在上一节中, 进行了 Qt 的 GUI 测试, 如果需要一系列不同的测试, 同批量数据测试一样, 可以通过记录一系列的 GUI 事件, 然后分别进行重放来完成多项测试。

这里还是使用上一节的 QSpinBox 测试, 但这次添加了不同的按键测试。

程序如下所示:

```
#include <QtGui>
#include <QtTest/QtTest>

class TestGui: public QObject
{
    Q_OBJECT

private slots:
    void testGui_data();
    void testGui();
};

void TestGui::testGui_data()
{
    QTest::addColumn<QTestEventList>("events");
    QTest::addColumn<int>("expected");

    QTestEventList list1;
    list1.addKeyClick(Qt::Key_Up);
    QTest::newRow("up") << list1 << 11;

    QTestEventList list2;
    list2.addKeyClick(Qt::Key_Down);
    QTest::newRow("down") << list2 << 9 ;
}

void TestGui::testGui()
{
    QFETCH(QTestEventList, events);
    QFETCH(int, expected);

    QSpinBox spinBox;
    spinBox.setMinimum(1);
    spinBox.setMaximum(100);
    spinBox.setSingleStep(1);
    spinBox.setValue(10);

    events.simulate(&spinBox);

    QCOMPARE(spinBox.value(), expected);
}
```



```
}  
  
QTEST_MAIN(TestGui)  
#include "guireplay.moc"
```

这里，使用 `QTestEventList` 作为一列记录要重放的事件，另外一列记录重放的结果。在测试中，使用 `events.simulate()` 函数重放相应的事件。需要注意，每次测试都是独立的，也就是说在这里每次测试前，`spinBox` 的初始值都是“10”。

测试结果如下：

```
***** Start testing of TestGui *****  
Config: Using QTest library 4.3.2, Qt 4.3.2  
PASS : TestGui::initTestCase()  
PASS : TestGui::testGui()  
PASS : TestGui::cleanupTestCase()  
Totals: 3 passed, 0 failed, 0 skipped  
***** Finished testing of TestGui *****
```

21.3 小 结

Qt 的单元测试框架提供了自动测试的手段。通过 `QTestLib`，能够进行批量数据测试和 GUI 测试，以便及早地检测出应用程序的问题。`QTestLib` 也为测试驱动开发提供了相应的手段。

附录 A Qt 安装

Qt 4.3 (以下简称 Qt) 可以在 Linux, Solaris, Windows, Mac OS X 等多种平台上安装。这里简要地介绍在前三个平台上的安装步骤。如不特别指明, 以下介绍的都是开源版的 Qt。

A.1 Linux 平台

Qt 在 Linux 上安装非常简单, 具体步骤如下:

1. 获取源代码

可以从 Trolltech 公司的 FTP 服务器下载, 地址是 <ftp://ftp.trolltech.com/qt/source>。在本书写作时, 最新的 Qt 源代码包是 `qt-x11-opensource-src-4.3.2.tar.gz`。该版本中的 Qt 设计器为中文界面。

2. 解压缩

将下载的源代码包解压缩到用户目录, 操作如下:

```
tar xvfz qt-x11-opensource-src-4.3.2.tar.gz
```

Qt 源码可以用普通用户身份编译, 但安装到默认的 `/usr/local/Trolltech` 目录时则需要 root 用户权限。

3. 生成 Makefile

运行 `configure` 生成 `Makfile` 文件, 如下:

```
./configure
```

运行后出现如下提示:

```
This is the Qt/X11 Open Source Edition.
```

```
You are licensed to use this software under the terms of either  
the Q Public License (QPL) or the GNU General Public License (GPL).
```

```
Type 'Q' to view the Q Public License.  
Type 'G' to view the GNU General Public License.  
Type 'yes' to accept this license offer.  
Type 'no' to decline this license offer.
```

Qt 开源版可以使用 QPL 或 GPL 协议, 输入 `yes` 接受协议即可开始配置。之后显示 Qt 检测到的配置, 在作者的 OpenSuSE Linux 10.2 上显示的配置如下:

```
Build type:    linux-g++  
Architecture: i386
```



Platform notes:

- Also available for Linux: linux-kcc linux-icc linux-cxx

Build libs tools examples demos

Configuration release shared dll largefile stl precompile_header
 separate_debug_info mmx 3dnow sse sse2 qt3support accessibility opengl minimal-config
 small-config medium-config large-config full-config reduce_exports ipv6 clock-monotonic
 mremap getaddrinfo ipv6ifname getifaddrs inotify png system-freetype system-zlib nis cups
 iconv glib qdbus openssl x11sm xshape xinerama xcursor xfixes xrandr xrender fontconfig
 tablet xkb release

Debug no
 Qt 3 compatibility .. yes
 QtDBus module yes
 STL support yes
 PCH support yes
 MMX/3DNOW/SSE/SSE2.. yes/yes/yes/yes
 IPv6 support yes
 IPv6 ifname support . yes
 getaddrinfo support . yes
 getifaddrs support .. yes
 Accessibility yes
 NIS support yes
 CUPS support yes
 Iconv support yes
 Glib support yes
 Large File support .. yes
 GIF support plugin
 TIFF support plugin (qt)
 JPEG support plugin (qt)
 PNG support yes (qt)
 MNG support plugin (qt)
 zlib support system
 OpenGL support yes
 NAS sound support ... no
 Session management .. yes
 XShape support yes
 Xinerama support yes
 Xcursor support yes
 Xfixes support yes
 Xrandr support yes
 Xrender support yes
 FontConfig support .. yes
 XKB Support yes
 immodule support yes
 SQLite support plugin (qt)
 OpenSSL support yes

如果有一些选项显示为 `no`，一般是 Qt 没有检测到相应的库或头文件。安装相应软件包的库和头文件即可。如果需要手动选择 Qt 的配置选项，可以输入：

```
./configure --help
```

来查看 Qt 的选项，其中带“*”号的选项是默认选项，“+”号代表需要检测成功才使用的默认选项。如更改选项，只需要在配置时加入自己的选项。例如需要编译 `mysql` 数据库驱动可以键入：

```
./configure -plugin-sql-mysql
```

但成功与否取决于 Linux 系统上是否有 `mysql` 数据库的相应头文件和库。

如果配置失败，可以在配置时使用“-v”选项查看具体出错原因，如下：

```
./configure -v
```

如果需要在第一次配置的基础上改变配置选项，通常需要运行“`make confclean`”清除之前的配置。

4. 编译

配置成功后，直接键入“`make`”即可编译。编译时间视具体机器配置不同而不同，在作者的双核 P4 3.0，1G 内存的机器上，需要不到 1 小时的时间。

5. 安装

编译完成后，直接输入：

```
make install
```

就可以将 Qt 安装到默认的 `/usr/local/Trolltech/Qt-4.3.2` 目录中。在编译完后，源代码不要删除，在以后需要调试到源代码时还可以使用。

6. 设置环境变量

为了更方便地使用 Qt，可以在用户的 shell 启动文件中设置相应环境变量。如果用户使用的 shell 是 `bash`，则可以在 `.bash_profile` 中加入：

```
export QTDIR=/usr/local/Trolltech/Qt-4.3.2
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

如果有多台机器要装 Qt，则可以把第一台上编译好的 Qt 制作为 `rpm` 包，供其他机器直接使用。

A.2 Solaris 平台

Qt 安装在 Solaris 平台上要稍微复杂一些，因为 Solaris 缺少一些 Linux 上的 GNU 软件。这可以通过安装相应的软件来完成。Solaris 操作系统分为 SPARC 和 X86 两种版本，这里以 X86 版本为例介绍，主要讲解和 Linux 不同的地方。

在 Solaris 10 X86 版上安装需要很多 GNU 的软件包，可以到 <http://www.sunfreeware.com> 下载。包括：

```
binutils-2.17-sol10-x86-local.gz
make-3.81-sol10-x86-local.gz
```



```
gcc-3.4.5-sol10-x86-local.gz
libiconv-1.11-sol10-x86-local.gz
tar-1.16-sol10-x86-local.gz
```

这些软件包都是 Solaris 的 package 格式，安装方法如下（以 tar 为例）：

```
gzip -d tar-1.16-sol10-x86-local.gz
pkgadd -d tar-1.16-sol10-x86-local.gz
```

注意 pkgadd 命令必须以 root 身份运行。键入 pkgadd 后出现：

```
The following packages are available:
```

```
1  SMCTar      tar
      (x86) 1.16
```

```
Select package(s) you wish to process (or 'all' to process
all packages). (default: all) [?..??:q]:
```

直接回车（安装所有）就可以安装包了。

注意 Qt 的源代码包必须使用 GNU 的 tar 解开才能保证解压的完整性。

在 Solaris 10 上通常预装有 Sun Studio，所以在编译 Qt 时既可以使用 Sun Studio 的 CC 编译器，也可以使用 GNU g++ 编译器。要明确指定使用那种编译器，可以使用命令行参数或 QMAKESPEC 环境变量。如要使用 g++ 编译器，可以使用：

```
./configure --platform=solaris-g++
```

也可以运行：

```
export QMAKESPEC=solaris-g++
```

来强制指定使用 g++ 编译器。solaris-g++ 也可以替换为 solaris-cc、solaris-cc-64、solaris-g++-64，分别代表 32 位 CC 编译器，64 位 CC 编译器和 64 位 g++ 编译器。

安装完后，需要在用户的 .bash_profile（假设用户使用的是 shell 是 bash）中进行一些设置。加入的 shell 命令如下：

```
export QTDIR=/usr/local/Trolltech/Qt-4.3.2
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:/usr/openwin/sfw/lib:$LD_LIBRARY_PATH
```

上面最后一行中的 /usr/openwin/sfw/lib 是 libXrender.so 库路径，如果是 AMD 的 64 位 CPU 则改为 /usr/openwin/sfw/lib/amd64。

A.3 Windows 上安装

开源版的 Qt 在 Windows 操作系统上只能在 MinGW 上编译，而商业版的则可以和 Visual Studio 紧密集成。Trolltech 公司也正在开发和 Eclipse 集成的 Qt 开发插件。

MinGW 的安装

MinGW 是 Minimalist GNU for Windows 的缩写，是在 Windows 上的 GNU 应用工具集。在 Windows 上安装开源版 Qt，首先要安装 MinGW 环境。具体步骤如下：

1. 下载 MinGW

MinGW 可以在 <http://mingw.sourceforge.net> 下载, 本书写作时的最新版本是 MinGW-5.1.3.exe。下载后双击执行程序, 出现如图 A-1 所示欢迎窗口。

单击 Next 按钮进入下载界面, 如图 A-2 所示。

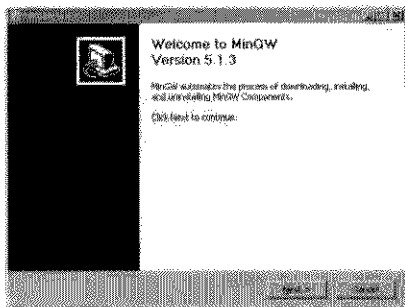


图 A-1 MinGW 欢迎窗口

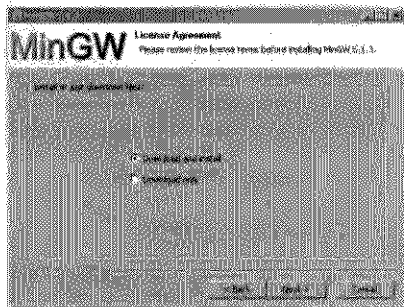


图 A-2 MinGW 下载界面

该窗口说明 MinGW 需要下载相应的组件, 单击 Next 按钮到下一步, 如图 A-3 所示。

这里要求阅读协议, 一些组件是基于 GPL 的, 一些是基于其他协议的。单击 “I Agree” 接受协议到下一步如图 A-4 所示。

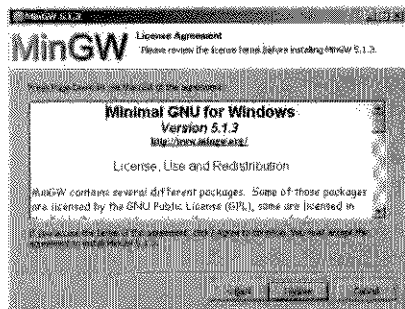


图 A-3 MinGW GPL 协议

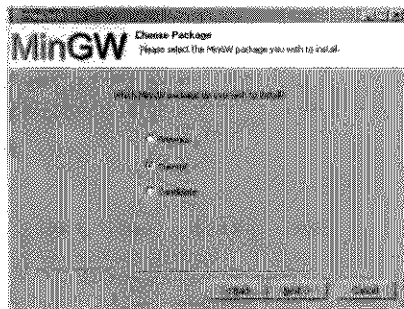


图 A-4 MinGW 安装包类型

在这个对话框中, 选择要安装的 MinGW 版本是以前的、当前的还是预发行的, 选择默认的 Current 即可就进入了下一步, 如图 A-5 所示。

在这一步, 选择要安装的 MinGW 包, 只需要选择 gcc、g++ compiler、MinGW Make 就可以了。接下来选择 MinGW 在系统菜单中的位置, 如图 A-6 所示。

可以直接使用默认值并单击 Next 后就开始下载并安装了, 最后安装成功的画面如图 A-7 所示。

如果要使用 gdb 进行调试, 还需要单独下载 gdb 并安装, 下载地址是 <http://downloads.sourceforge.net/mingw/gdb-6.6.tar.bz2>。下载后直接将文件解压缩到之前安装的 MinGW 目录并选择覆盖就可以了。

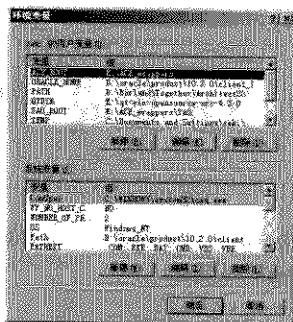


图 A-8 设置环境变量

在用户变量中设置 PATH 环境变量为：e:\qt-win-opensource-src-4.3.2;%PATH%。

在 Windows Vista 中，需要对 MinGW 进行额外的设置才能正常使用（在后续的版本中也许可以解决这个问题）。具体步骤如下：

01 设置 GCCPREFIX 环境变量为 MinGW 的安装目录，如：D:\MinGW；

02 设置 PATH 环境变量为 %GCCPREFIX%\libexec\gcc\mingw32\3.4.2;%PATH%，其中 3.4.2 根据实际的 gcc 版本可能不同；

03 编译 Qt 的步骤与在 Windows XP 上相同。

Windows 下的 Qt 商业版可以和 Visual Studio.NET 集成，使用非常方便，这里不做介绍。

附录 B Qt 集成开发环境

使用集成开发环境可以加快开发速度,改进调试手段。可以使用多种集成开发环境来开发 Qt 应用程序,这里只介绍常用的 KDevelop 和 Eclipse。

B.1 KDevelop

在 Linux 操作系统上,通常选择 KDevelop 进行开发。KDevelop 是基于 KDE 的集成开发环境,KDevelop 支持基于 C/C++, Java, Python, Ruby, Perl, PHP, SQL, C#, Fortran, bash, Pascal, Ada 等多种语言的软件开发。可以和 Subversion、CVS、ClearCase、perforce 等多种配置管理工具集成。此外还提供了二次开发接口,供用户扩展功能。由于 KDevelop 支持 Qt 应用程序的开发,所以在 Linux 上通常使用 KDevelop 来开发 Qt 应用程序。这里将简单地介绍如何安装 KDevelop。

目前 Linux 上自带的 KDevelop 版本都比较旧,主要是支持 Qt 3 应用程序的开发,对 Qt 4 的支持都不太好,所以推荐安装在本书写作时最新的 KDevelop 3.5.0。

KDevelop 的源代码可以从 <http://www.kdevelop.org> 下载。在这个网站上,提供一些流行的 Linux 平台的 KDevelop 的二进制版本,可以直接下载安装。这里只讲解如何从源代码编译。具体步骤如下。

1. 安装必备的软件

KDevelop 3.5 要求在 KDE 3.4 以上版本上安装,如果 KDE 版本低于 3.4(例如红旗 Linux Workstation 5.0),则必须将 KDE 升级到 3.4 以上。

由于从 KDevelop 3.4 版开始,KDevelop 和 gdb 的接口改为了 gdb/mi 接口,而不是 gdb/cli 接口,使得调试更为可靠,KDevelop 3.4 要求使用 gdb 6.6。大多数 Linux 发行版的 gdb 的版本都比 6.6 低,所以装 KDevelop 之前需要安装 gdb 6.6。

gdb 6.6 源代码包可以从 <http://sourceware.org/gdb> 下载,安装过程如下:

```
tar xvfj gdb-6.6.tar.bz2
cd gdb-6.6
./configure
make
make install
```

其中最后一步必须以 root 身份运行。

KDevelop 的安装还需要 flex、Berkeley DB、graphviz 等软件包。如果 Linux 上没有预装这些软件,则需要进行安装。如果需要 KDevelop 集成 Subversion 配置管理工具(缺省配置包括该集成选项),还需要安装 Subversion 的开发软件包。

2. 在用户目录解压缩

```
tar xvfz kdevelop-3.5.0.tar.bz2
```

3. 生成 Makefile 文件

在终端窗口中运行：

```
cd kdevelop-3.5.0
./configure
```

即可生成 Makefile 文件。因为 KDevelop 是基于 Qt 3 开发的，如果用户已经安装了 Qt 4，则需要键入：

```
./configure --with-qt-dir=/usr/lib/qt3
```

如果用户的 Qt 3 目录不在 `/usr/lib/qt3` 下，则改成实际的目录。如果安装过程中缺少某些软件包，可以安装缺少的软件包后重新运行 `configure`。

4. 编译

键入“make”就可以编译了，这个时间视硬件配置不同而异，一般需要一个小时左右。

5. 安装

以 root 身份输入：

```
make install
```

就可以安装 KDevelop 了。

6. 运行

可以用菜单或在终端窗口直接输入“kdevelop”运行 KDevelop。KDevelop 的界面如图 B-1 所示。

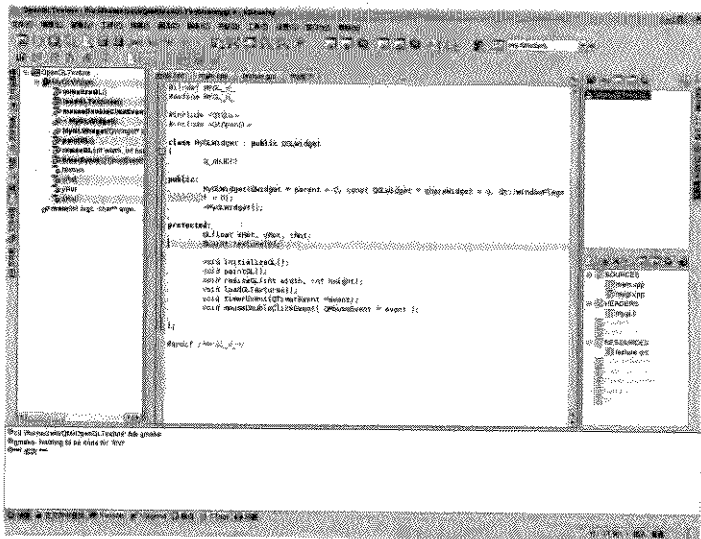


图 B-1 KDevelop 3.5 界面



7. 中文输入问题

在 KDevelop 中使用 SCIM 输入汉字会出现汉字在屏幕上只能保持一秒钟左右就迅速消失的情况，根本来不及选字，这是 KDevelop 的一个 Bug。在 KDevelop 3.3 版中只需要修改 QT_IM_MODULE 就能解决这个问题。以 OpenSUSE 10.2 为例，以 root 身份编辑文件/etc/X11/xim.d/scim，找到：

```
export QT_IM_MODULE=scim
```

将其改为：

```
export QT_IM_MODULE=xim
```

其他 Linux 系统文件中 SCIM 位置可能不一样，如果找不到，直接在用户的 .bash_profile 文件（假定使用的 Shell 是 bash）中添加这一行也可以。

在 KDevelop 3.5 中除了要修改 SCIM 文件之外，还需要对 Qt 3 进行配置。因为 KDevelop 3.5 是使用 Qt 3 开发的，所以一定要配置 Qt 3 而不是 Qt 4。运行/usr/lib/qt3/bin/qtconfig（不同的 Linux 位置可能不一样），如图 B-2 所示。

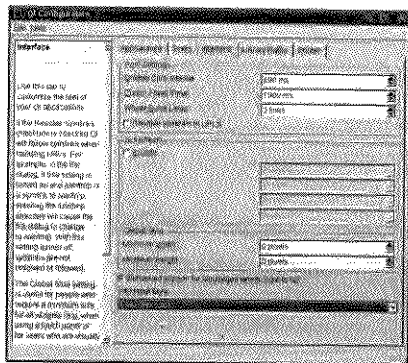


图 B-2 Qt3 配置工具

选择 Interface 标签，将 XIM Input Style 从默认的“On The Spot”改为“Over The Spot”，就可以正常输入汉字了。原来的“On The Spot”风格是指输入法预编辑区域由客户程序显示，也就是在 KDevelop 的编辑器显示。“Over The Spot”又称光标跟随风格，就是输入预编辑区域和候选区在一个窗口，随着输入位置的改变而移动，如图 B-3 所示。

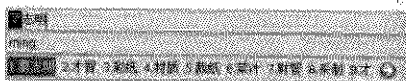


图 B-3 Over The Spot 模式的 SCIM

需要注意的是，如果使用的是 root 用户运行 qtconfig，修改结果不能保存。如果用户一定要在 root 用户下使用 KDevelop，可以将普通用户目录下的 .qt 目录下的配置好的 qtrc 文件（qtconfig 的配置结果在 qtrc 文件中）复制到 root 用户目录下的 .qt 目录。

修改完成后，重新启动 KDevelop 就能输入汉字了。

8. 文件编码

默认情况下 KDevelop 的文本编辑器默认编码是 KDE 的默认值 (OpenSuSE 10.2 中是 UTF-8)。如果 KDevelop 编辑的程序还需要在 Windows 下使用, 就需要使用一致的编码。这可以在 KDevelop 的“设置→配置编辑器”菜单弹出的对话框中进行配置, 如图 B-4 所示。

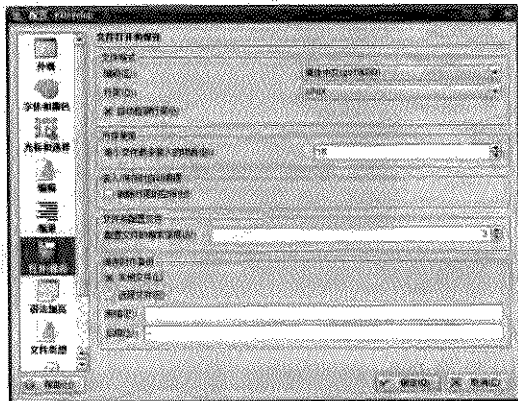


图 B-4 KDevelop 编码设置

点击对话框左边的“打开/保存”按钮, 然后选择右边面板中的编码, 这里选择了简体中文 (gb18030)。也可以选择 UTF-8 等其他可以保存中文的编码, 只要和 Windows 中的其他开发工具保持一致就可以了。

B.2 Eclipse

在 Windows 平台上没有 KDevelop, 可以使用开源的 Eclipse+CDT 来开发 Qt 应用程序。Eclipse+CDT 不能解析 qmake 工程文件, 但具有跨平台的好处, 同一个工程可以在 Linux、Solaris、Windows 等多种平台下使用。目前 Trolltech 公司也正在开发 Qt 的 Eclipse 插件, 在本书写作时, 还只有 1.0rc2 版。这里以 Eclipse Europa (Eclipse 3.3+CDT 4.0.1) 在 Windows 平台上用开源版 Qt 为例, 说明如何使用 Eclipse 开发 Qt 应用程序。

使用 Eclipse CDT 之前需要进行一些配置以适应 Qt 的开发。包括:

1. 设置默认的工程属性

选择 Eclipse 的菜单“Windows”|“Preferences”, 出现设置窗口如图 B-5 所示。

在窗口中选择“C/C++”→“New CDT Project”→“Makefile Project”选项, 如图 B-5 所示。在右边的“Builder settings”标签下将 Build command 改为 mingw32-make。在“Behaviour”标签下根据需要编译的版本的不同, 在 Build(Incremental Build)中可根据需要填入空白(即默认的 debug 版), release (release 版)或 all (Debug 和 Release 版)。如图 B-6 所示。

还需要在“Binary Parser”标签下选择“PE Windows Parser”, 这样 CDT 才能认出 MinGW 编译的执行文件。如图 B-7 所示。

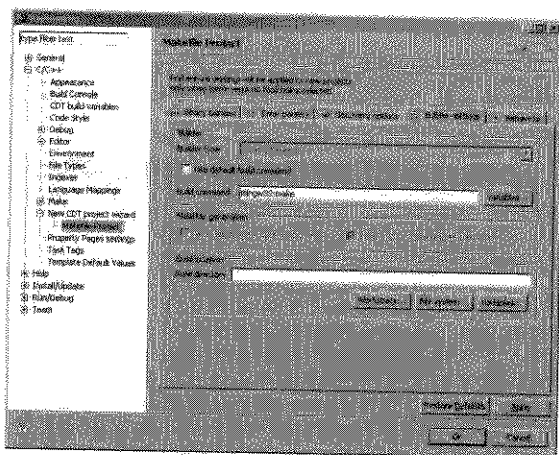


图 B-5 Makefile Project 的 Build Command 设置

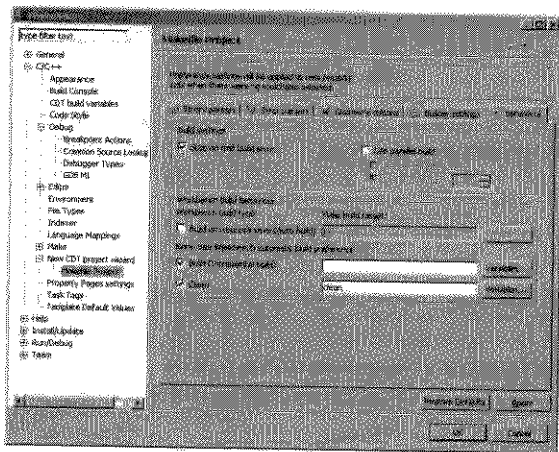


图 B-6 Makefile Project 的 Behaviour 设置

这些设置只对没有选择 toolchain 的新建工程时起作用。如果选择了 MinGW GCC 的 toolchain, 则还需要在新建工程时单独设置。

2. 其他设置

在“Windows→Preferences”对话框中, 选择“General”选项, 选择“Always run in background”, 可以使 make 过程总是在后台运行, 而不用每次手动将 make 置于后台运行。

为了在 Eclipse 中使用 qmake, 可以依次选择“Run→External Tools→External Tools”菜单, 如图 B-8 所示。

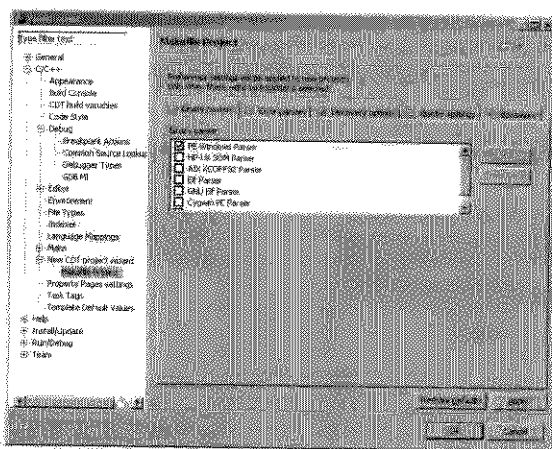


图 B-7 Makefile Project 的 Binary parsers 设置

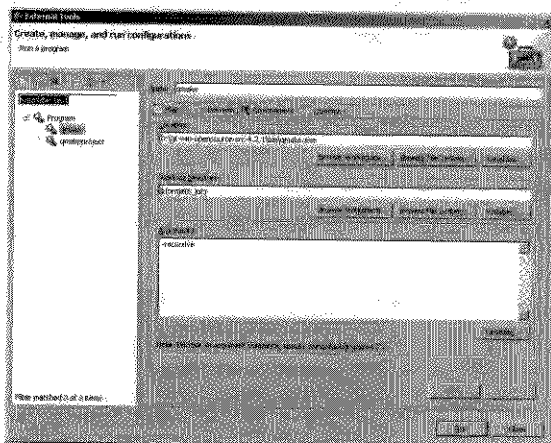


图 B-8 qmake 设置

在对话框中，建立一个 qmake 的外部工具。这里有三项需要输入。“Name”栏是外部工具名称，可以输入 QMAKE。“Location”文本框是工具的执行文件名，可以直接输入 qmake 的完整路径，或点选“Browse File System”按钮找到 qmake 文件。“Working Directory”框中可以直接输入“\${project_loc}”，或点选“Variables”按钮选取“project_loc”变量，该变量是选中的资源所在工程的绝对路径，也就是 qmake 的工作路径。最后在“Arguments”栏中输入“-recursive”参数，这样 qmake 就可以自动在工程的所有子目录中运行了。设置完毕后，可以在菜单和工具栏中直接调用。

同样可以建立 designer、assistant、linguist、lupdate、lrelease 等外部工具。

为了让.ui 文件在 Eclipse 中能自动打开，可以在 Windows 的资源管理器中建立 ui 文件和 Qt 设计器的关联。然后在 Eclipse 双击.ui 文件，就可以使用 Qt 设计器打开该 ui 文件了。



进行了上面的配置后，可以开始使用 Eclipse-CDT 开发 Qt 程序了。建立工程时应该选择“C++ Project”，接下来的对话框中“Project types”选择“Makefile Project”，“Toolchain”项选择“MinGW GCC”（也可以选择“other toolchain”，这样就可以使用全局设置的 Build 属性了），如图 B-9 所示。

单击 Next 进入下一个对话框，如图 B-10 所示。

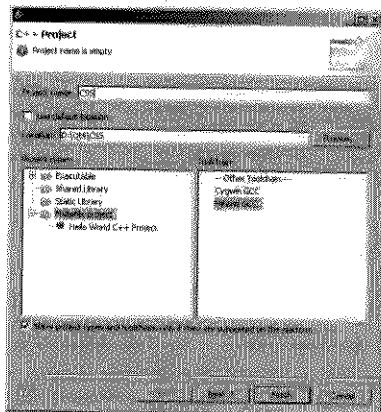


图 B-9 建立 Makefile project

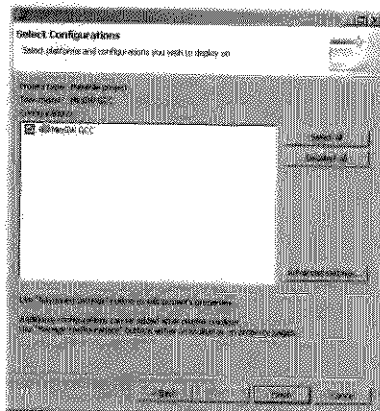


图 B-10 设置 Tool-chain

在这里还需要单击“Advanced settings”按钮进行设置，如图 B-11 所示。

在这个对话框中在右边面板“Builder settings”中将“Build command”改为“mingw32-make”（如果觉得每次这样改很麻烦，也可以将 MinGW 的 mingw32-make.exe 更名为 make.exe）。将“Behaviour”标签下的“Build”（Increment Build）中的“all”删除。然后单击“OK”确认修改。接下来单击“Finish”就可以建立新工程。

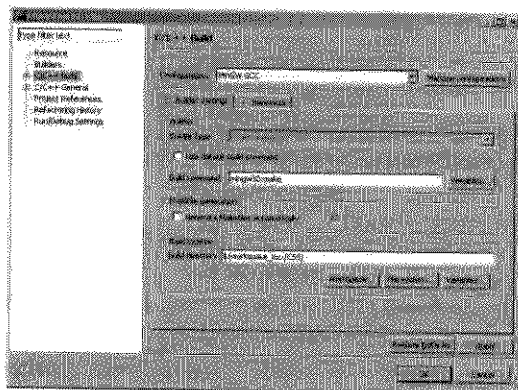


图 B-11 设置 MinGW 的 Build command

如果需要自己创建 make target，在工程上单击鼠标右键，选择“Make targets→Create...”来创建 release 或 debug 的 target。图 B-12 显示了建立了 debug 的 target。

要运行程序需要生成一个运行配置。选择“Run→Open Run Dialog”菜单一次性设定或直接单击运行按钮按提示一步步设定,选择菜单后出现的对话框如图 B-12 所示。

在对话框中增加一个“C/C++ Local Application”,如需使用命令行参数则在“Arguments”标签中填入程序参数。如需调试则要安装 MinGW 版的 gdb。

需要注意在 Windows 下,Eclipse 中文本文件的默认编码是 GB18030, Linux 下默认编码大多是 UTF-8, KDevelop 的默认编码也是 UTF-8。因此在 Windows 下的编辑的文件在 Linux 下汉字将不能正常显示。可以在“Window→Preferences”菜单下修改文本文件的默认编码,在弹出的对话框中选择“General→Workspace”,然后在右边面板中的“Text file encoding”中选择“Other”中的“UTF-8”编码。如图 B-14 所示。

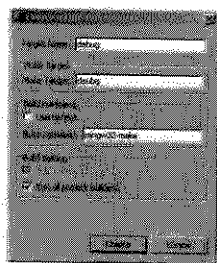


图 B-12 创建 Make target

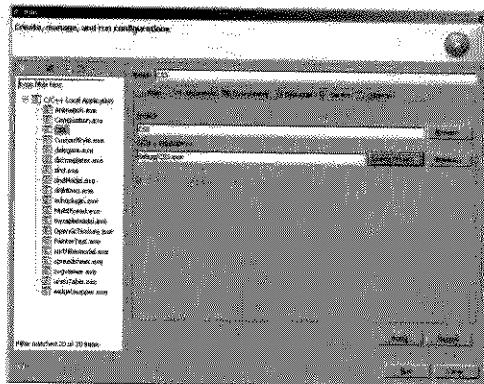


图 B-13 设置运行配置

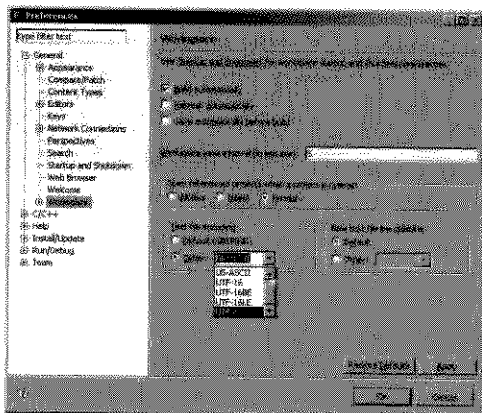


图 B-14 设置全局文本文件编码

对于单独的文件设置编码则是选择“Edit-Set Encoding”菜单，弹出对话框如图 B-15 所示，选择需要使用的编码即可。



图 B-15 设置单个文件的编码

codeForName(“UTF-8”))。

Trolltech 公司开发的 Qt Eclipse 插件可以集成 Qt 设计器、资源文件管理、make 工程管理、Assistant 等功能，使用起来非常方便。在本书完成时该软件还未推出正式版，就不详细说明了，有兴趣的读者可以自行试用。图 B-16 显示了在 Eclipse 中使用 Qt 设计器的界面。

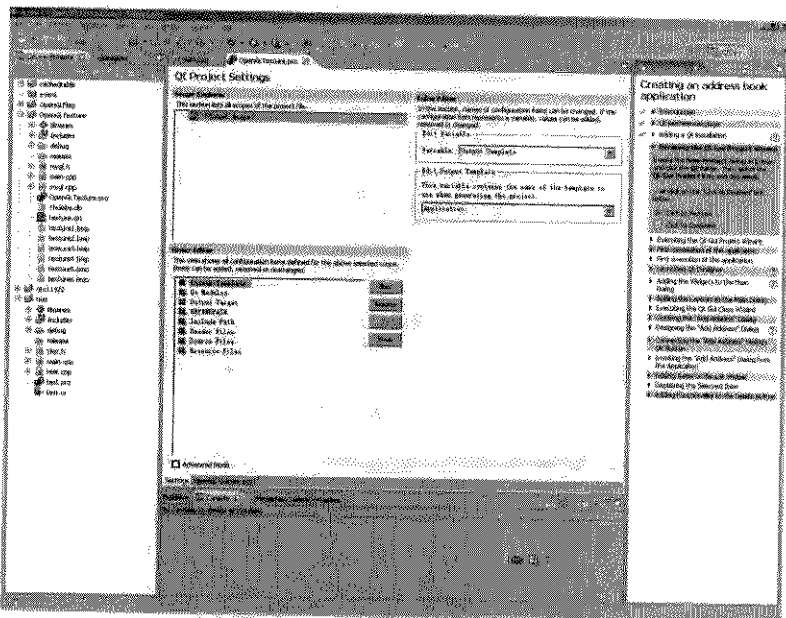


图 B-16 Trolltech 公司的 Qt Eclipse Integration

附录 C qmake 速查

qmake 是一种跨平台的工程文件形式,它类似于 cmake、automake 和 MPC 之类的工具。使用 qmake 可以获得在不同平台下使用相同的工程文件的好处,而且非 Qt 应用程序中也可以使用 qmake。本附录介绍了 qmake 开源版的一些比较常见的用法,更详细的内容可以参考 qmake 手册。

C.1 工程模板

qmake 根据生成的目标程序的不同,将工程文件分成了集中模板,包括 app、lib 等模板。

1. 应用程序模板

应用程序模板生成的 Makefile 将构建可执行程序。使用应用程序模板需在工程文件中加入:

```
template = app
```

应用程序模板中常用的 qmake 系统变量如表 C-1 所示。

表 C-1 应用程序模板的 qmake 系统变量

qmake 变量	含 义
HEADER	头文件列表
SOURCES	源文件列表
FORMS	ui 文件列表
TARGET	最终可执行的目标文件名。默认情况下目标主文件名和工程文件主文件名同名,扩展名根据取决于操作系统决定
DESTDIR	生成的最终目标文件的路径
DEFINES	应用需要的预处理定义
INCLUDEPATH	额外的头文件路径
DEPENDPATH	应用程序依赖文件搜索路径
VPATH	提供的文件搜索路径

2. 库模板

如果要生成共享库(动态链接库),可以使用库模板。在工程文件中的语法是:

```
template = lib
```

库的类型由 CONFIG 变量决定,dll 表示共享库(动态库),staticlib 表示静态库,plugin 表示插件(也是动态库的一种)。库模板中除了可以使用应用程序模板中的一些变量外,还可以使用 VERSION 变量来指定库的版本。



3. subdirs 模板

subdirs 模板中每个子目录都有独立的工程文件，并且是主工程的一部分。如：

```
SUBDIRS = graphics \
        map
```

需要注意每个子目录下的工程文件主文件名必须和目录名同名。如果要求子目录按提供的顺序编译，则需要修改 CONFIG 变量的值如下：

```
CONFIG += ordered
```

C.2 常见任务

1. 使用预编译头文件

将稳定的代码进行预编译可以加快编译速度。要使用预编译头文件，需要在工程中加入：

```
CONFIG += precompiled_header
```

然后定义需要预编译的头文件：

```
PRECOMPILED_HEADER = mypch.h
```

需要注意预编译头文件中 C 和 C++ 的代码应该分开，如下所示。

// 以下是 C 的头文件

```
.....
#ifdef __cplusplus
// 以下是 C++ 的头文件
#include <iostream>
#include <vector>
#include <QApplication>
#include <QMainWindow>
#include <QWidget>
#include "my_graphics.h"
...
#endif
```

2. 将临时文件生成到指定目录

在默认情况下，临时文件（moc 文件、.o 目标文件等）和源文件在同一个目录，不便管理。可以使用 MOC_DIR、RCC_DIR、UI_DIR、OBJECTS_DIR 等指定 moc、rcc、uic、gcc/g++ 产生的中间文件位置，如：

```
MOC_DIR = tmp/moc
RCC_DIR = tmp/rcc
UI_DIR = tmp/ui
OBJECTS_DIR = tmp/obj
```

3. 使用操作符

qmake 中有一些操作符，可以对变量赋值。最简单的是“=”操作符，表示直接给变量赋值。还有

“+=”表示追加一个新值到变量上。“-=”则是在变量的值列表中去掉指定的值。“*=”表示只有指定的值在变量中不存在时，才需要添加。“~=”类似于 vim 中的替换操作，它将匹配正则表达式的值替换为新值。如：

```
HEADERS ~= s/czm/zeki
```

上面的语句将 HEADERS 中所有的“czm”字符替换为“zeki”。

4. 获取各种变量的值

获取 qmake 变量的值使用“\$\$”，如：

```
ALL_DEFINES = $$DEFINES
message("工程中的所有定义")
message($$ALL_DEFINES)
```

也可以使用“\${DEFINES}”的形式，这种方式通常在连接操作时使用。

获取环境变量使用“\$(ENVIRONMENT_VAR)”的形式，如：

```
LIBS += -L$(ORACLE_HOME)/lib/clntsh
```

“\${...}”用来存取配置选项，如：

```
${QT_INSTALL_PLUGINS}
```

用来获取 Qt 的插件安装目录。其他配置变量还有：QT_VERSION 返回 Qt 版本，QT_INSTALL_PREFIX 返回 Qt 的安装目录等。

5. 生成 debug 和 release 版

要同时生成工程的 debug 和 release 版，可以使用如下的形式：

```
CONFIG += debug_and_release
CONFIG(debug, debug|release) {
    unix: TARGET = $$join(TARGET,,_debug)
    win32: TARGET = $$join(TARGET,,d)
} else {
    TARGET = myapp
}
```

第一行表示要生成 debug 和 release 版的执行文件。第二行告诉在 Windows 下 debug 版执行文件添加后缀“d”，Linux 下的 debug 版文件添加后缀“_debug”。生成 Makefile 之后，要编译 debug 版文件，则输入：

```
make debug
```

要生成 release 版执行文件，输入：

```
make release
```

两种版本都生成，输入：

```
make all
```

如果 make 不带参数，则生成 debug 版文件。



6. 添加额外的库和头文件

如果在应用中需要使用其他的库和头文件，可以用如下的形式：

```
LIBS += -L/usr/local/lib -lsubversion
INCLUDEPATH = /usr/local/include/subversion
```

上面的 LIBS 变量中“-L”指明的库的搜索路径，“-l”指明了库名。通常在 Linux 下库的实际文件名应该是 libsubversion.so.3 这样的形式，但只用指定 lib 和.so 中间的库名就可以了。

INCLUDEPATH 变量添加了头文件的搜索路径。

7. 编译警告选项

如果需要关闭编译警告，使用：

```
CONFIG += warn_off
```

打开编译警告，使用：

```
CONFIG += warn_on
```

8. 使用范围和条件

范围 (Scope) 是指满足条件后要处理的 qmake 语句块。例如：

```
unix {
    SOURCES += unixspec.cpp
}
```

上面的代码表明在 UNIX 平台上加入特定的源文件 unixspec.cpp。条件也可以使用“!”操作符，表示不满足条件则处理相应的指令。如：

```
! unix {
    SOURCE += otherspec.cpp
}
```

条件可以嵌套，如：

```
unix {
    debug {
        TARGET = mytarget_debug
    }
}
```

也可以简写为：

```
unix:debug: TARGET += mytarget_debug
```

条件判断中还可以使用 else 语句，如：

```
unix:opengl {
    SOURCES += myopengl.cpp
} else:unix {
    SOURCES += unixnoopengl.cpp
} else {
```



```
SOURCES += neopengl.cpp
}
```

可以用作范围的有 CONFIG 变量中的值、QMAKESPC 变量，如：

```
unix-cc {
    message("using sun cc compiler!")
}
```

9. 使用 make install 进行安装

在 Linux 下编译安装源代码包，通常是在编译后运行 “make install” 来进行安装。在 Qt 中也可以实现这个功能。qmake 中的 INSTALLS 变量定义了在执行 “make install” 时需要安装的文件。要安装每个条目都定义了一些安装信息，如安装的路径，如下所示。

```
target.path = /usr/local/bin
sources.files = $$HEADERS
sources.path = /usr/local/include
INSTALLS += target sources
```

上面的代码定义了 target、sources 表示要安装目标文件和源文件，每个安装项目有安装路径和文件。其中 target 就是最终编译出来的执行文件。

10. 生成分发文件

在 UNIX (Linux) 环境下，可以使用 “make dist” 生成一个压缩的源代码分发。如果要添加要分发的文件可以使用：

```
DESTFILES += README
```

默认情况下，Linux 下使用 gzip 压缩，MinGW 下使用 zip 压缩。

11. qmake 函数

qmake 提供了很多处理变量的函数，如 basename()、joining()、contains()、message() 等。具体使用方法可以查看 qmake 的手册。

附录 D 深入 Qt 源代码

同所有的软件一样，Qt 也不是完美无缺的。在使用中，可能会发现一些小 Bug。幸运的是，Qt 是开放源代码的软件，可以阅读其源代码，定位其中的问题。下面以 Qt 中的一个 Bug 为例，来简单介绍如何通过源代码查找问题。

这里分析的是 `hasPendingEvents()` 事件的问题。如将 Qt 的定时时间设为 0，则是在系统空闲时产生 `timerEvent()` 事件，也就是说这时候已经没有任何需要处理的窗口事件了。而事实上，在 Linux 系统上，`hasPendingEvents()` 总是返回“true”，也就是说系统总是忙！在 Windows 系统上也有这个问题。如果在 Linux 下设置环境变量：

```
export QT_NO_GLIB=1
```

`hasPendingEvents()` 表现就是正常的。这个原因是因为从 Qt 4.2 开始，底层的事件处理机制引入了 GTK+ 的底层库 GLib 为默认的事件分发器，而这个处理方法在 Qt 中的实现还有 Bug。如果设置了 `QT_NO_GLIB` 则使用传统的 UNIX 事件处理机制，就没有问题。尽管在 Qt 4.2.2 的 Changes 文件中有如下的说明：

Fixed QApplication::hasPendingEvents() returning true even if no events were pending when using the Glib event dispatcher.

但实际测试表明，这个 Bug 根本没有解决。这时可以通过分析 Qt 的源代码来看这个问题到底出在哪里。

Qt 库的源代码在源代码树中的 `src` 目录下，包括 `3rdparty`、`corelib`、`gui`、`network`、`opengl`、`plugins` 等目录。`3rdparty` 中是一些第三方软件，如 `des`、`freetype`、`libjpeg`、`sqlite` 等；`corelib` 中是 Qt 的核心库；其他的目录则是 Qt 的各个模块，如 `gui` 是 `QtGui` 模块。

Qt 的类实现通常都对应了一个私有类，同时使用宏 `Q_D()` 来定义，并使用变量 `d` 来引用。Qt 的头文件如果有 `_p` 后缀，如 `qaction_p.h`，表明该类是内部使用的，用户不可见。

Qt 源代码中出现很多的两个宏定义如下（`corelib/global/qglobal.h`）：

```
#define Q_D(Class) Class##Private * const d = d_func()
#define Q_Q(Class) Class * const q = q_func()
```

分析源代码时，拥有一个源代码浏览工具和调试器将会非常方便。在 Windows 下，可以使用 Eclipse，Linux 下则可以使用 KDevelop。

下面来逐步定位 `hasPendingEvents()` 问题。

`hasPendingEvents()` 是 `QCoreApplication` 类的一个静态函数，它的实现在 `corelib/kernel/qcoreapplication.cpp` 中，代码如下：

```
bool QCoreApplication::hasPendingEvents()
{
```

```

QAbstractEventDispatcher *eventDispatcher =
    QAbstractEventDispatcher::instance();
if (eventDispatcher)
    return eventDispatcher->hasPendingEvents();
return false;
}

```

可以看到函数最终调用的是 `QAbstractEventDispatcher` 类的 `hasPendingEvents()` 函数。但 `QAbstractEventDispatcher` 类只是一个抽象基类，这里要找出真正的 `EventDispatch` 类是什么。其实找出是哪个类的方法最简单的就是用调试器跟踪。

可以看到在 `QCoreApplication` 的 `init()` 初始化函数中有一段代码：

```

if (!QCoreApplicationPrivate::eventDispatcher)
    QCoreApplicationPrivate::eventDispatcher =
        d->threadData->eventDispatcher;
// otherwise we create one
if (!QCoreApplicationPrivate::eventDispatcher)
    d->createEventDispatcher();

```

上面的代码创建了事件分发类，实际上的代码是 `QCoreApplication::createEventDispatcher()` 创建事件分发管理对象。

```

void QCoreApplicationPrivate::createEventDispatcher()
{
    Q_Q(QCoreApplication);
    #if defined(Q_OS_UNIX)
    # if !defined(QT_NO_GLIB)
        if (qgetenv("QT_NO_GLIB").isEmpty() &&
            QEventDispatcherGlib::versionSupported())
            eventDispatcher = new QEventDispatcherGlib(q);
        else
    # endif
        eventDispatcher = new QEventDispatcherUNIX(q);
    #elif defined(Q_OS_WIN)
        eventDispatcher = new QEventDispatcherWin32(q);
    #else
    # error "QEventDispatcher not yet ported to this platform"
    #endif
}

```

可以看出，在 UNIX (Linux) 环境下，有 `QEventDispatcherGlib` 和 `QEventDispatcherUNIX` 两个类。Windows 下则使用 `QEventDispatcherWin32` 类。

接下来就可以进入 `corelib/kernel/qeventdispatcher_glib.cpp` 看一下 `hasPendingEvents()` 函数的实现了。

```

bool QEventDispatcherGlib::hasPendingEvents()
{
    Q_D(QEventDispatcherGlib);
    return g_main_context_pending(d->mainContext);
}

```

该函数中调用了 GLib 事件循环中的函数 `g_main_context_pending()`，用来检查在指定的上下文中是否还有事件等待处理。GLib 是 GTK+ 和 GNOME 的底层核心库，提供了一些如事件循环、线程、动态加载、对象系统等功能。这里需要简单地说明一下 GLib 的事件循环机制。

GLib 的事件循环负责管理 GLib 和 GTK+ 应用中所有的事件源。事件源可以是文件描述符（如普通文件、管道和套接字）、定时事件和空闲事件等多种类型。新的事件源使用 `g_source_add()` 函数加入到事件循环。

GMainLoop 数据结构表示了主事件循环，它由 `g_main_new()` 函数创建。初始化事件源后，`g_main_run()` 函数开始事件循环，它不停地检查新的事件并将事件分发出去。最终退出事件循环可以由退出事件调用 `g_main_quit()` 来完成。Qt 的 GLib 事件分发就是基于 GLib 的事件循环机制的。

在 Qt 中，事件分为三类：GUI 事件、定时事件、Socket 事件。事件的分发分别使用 `postEventSourceDispatch()`、`timerSourceDispatch()`、`socketNotifierSourceDispatch()` 函数完成，这些函数将成为 GLib 事件循环中的回调函数使用。

每种事件要注册三个回调函数，分别是 `prepare`、`check` 和 `dispatch`。例如定时事件的三个回调函数分别是 `timerSourcePrepare()`、`timerSourceCheck()`、`timerSourceDispatch()`。首先来看一下 `timerSourceDispatch()` 函数，它在处理事件时调用。

```
static gboolean timerSourceDispatch(GSource *source, GSourceFunc, gpointer)
{
    (void) reinterpret_cast<QTimerSource *>
        (source)->timerList.activateTimers();
    return true; // ??? don't remove, right again?
}
```

Qt 中，定时事件使用 `QTimerInfoList` 的链表存储，`activateTimer()` 函数负责将到时的事件发送出去。

```
int QTimerInfoList::activateTimers()
{
    if (qt_disable_lowpriority_timers || isEmpty())
        return 0; // nothing to do

    bool firstTime = true;
    timeval currentTime;
    int n_act = 0, maxCount = count();

    QTimerInfo *saveFirstTimerInfo = firstTimerInfo;
    QTimerInfo *saveCurrentTimerInfo = currentTimerInfo;
    firstTimerInfo = currentTimerInfo = 0;

    while (maxCount-- > 0) {
        currentTime = updateCurrentTime();
        if (firstTime) {
            repairTimersIfNeeded();
            firstTime = false;
        }

        if (!isEmpty())
```

```

        break;

        currentTimerInfo = first();
        if (currentTime < currentTimerInfo->timeout)
            break; // no timer has expired

        if (!firstTimerInfo) {
            firstTimerInfo = currentTimerInfo;
        } else if (firstTimerInfo == currentTimerInfo) {
            // avoid sending the same timer multiple times
            break;
        } else if (currentTimerInfo->interval < firstTimerInfo->interval
            || currentTimerInfo->interval == firstTimerInfo->interval) {
            firstTimerInfo = currentTimerInfo;
        }

        // remove from list
        removeFirst();

        // determine next timeout time
        currentTimerInfo->timeout += currentTimerInfo->interval;
        if (currentTimerInfo->timeout < currentTime)
            currentTimerInfo->timeout = currentTime +
                currentTimerInfo->interval;

        // reinsert timer
        timerInsert(currentTimerInfo);
        if (currentTimerInfo->interval.tv_usec > 0 ||
            currentTimerInfo->interval.tv_sec > 0)
            n_act++;

        if (!currentTimerInfo->inTimerEvent) {
            // send event, but don't allow it to recurse
            currentTimerInfo->inTimerEvent = true;

            QTimerEvent e(currentTimerInfo->id);
            QCoreApplication::sendEvent(currentTimerInfo->obj, &e);

            if (currentTimerInfo)
                currentTimerInfo->inTimerEvent = false;
        }
    }

    firstTimerInfo = saveFirstTimerInfo;
    currentTimerInfo = saveCurrentTimerInfo;

    return n_act;
}

```



可以看到，在上面的函数中，定时事件每次都要重新计算下次发出的事件并重新插入到队列中。接下来就可以看检查当前是否有需要处理的定时事件函数。

```
static gboolean timerSourceCheck(GSource *source)
{
    GTimerSource *src = reinterpret_cast<GTimerSource *>{source};

    if (src->timerList.isEmpty())
        return false;

    if (src->timerList.updateCurrentTime() <
        src->timerList.first()->timeout)
        return false;

    return true;
}
```

从代码中可以看出，定时间隔为 0 的事件总是导致当前的定时队列中有需要处理的事件，所以上面的函数在这种情况下总是返回“false”。至此，hasPendingEvents()在空闲事件响应函数中总是返回“true”应该是真相大白了。虽然问题定位了，但要想解决这个问题还是比较复杂的，需要对 Qt 和 GLib 的事件处理机制有深入的了解，有兴趣的读者可以自己分析一下。当然最简单的办法还是使用：

```
export QT_NO_GLIB=1
```

来避免使用 GLib 的事件处理函数。

在 Windows 平台上也存在同样的问题（仅 Qt 4.3 中有问题，Qt 4.2 中没有此问题），下面也来分析一下原因。在 Qt 中事件循环是在 QApplication::exec()函数中建立的，实际的事件循环处理在 QEventLoop::exec()函数中进行。在 QEventLoop::exec()函数中，调用 processEvents()函数进行事件处理，如下：

```
try {
    while (!d->exit)
        processEvents(flags | WaitForMoreEvents |
            ProcessEventsFlag(QEventLoop::DeferredDeletion));
} catch (...) {
    .....
    throw;
}
```

在 Windows 平台上，QEventLoop::processEvents()函数实际上最终调用了 QEvent DispatchWin32::processEvent()函数，在该函数中对于每个 post 事件调用 QCoreApplication::sendPostedEvents()函数，如下：

```
do {
    QCoreApplication::sendPostedEvents(0,
        (flags & QEventLoop::DeferredDeletion) ? -1 : 0);
    .....
} while (canWait);
```

QCoreApplication::sendPostedEvents()实质上调用了 QCoreApplicationPrivate::sendPostedEvents()来发送 post 事件。

```
void QCoreApplicationPrivate::sendPostedEvents(QObject *receiver,
        int event_type, QThreadData *data)
{
    .....
    QCoreApplication::sendEvent(r, e);
    .....
    if (!data->postEventList.recursion && !event_type && !receiver) {
        const QPostEventList::iterator it = data->postEventList.begin();
        data->postEventList.erase(it, it + 1);
    }
}
```

在函数中，可以看到事件使用 sendEvent()函数发送出去，在这里，事件处理函数真正得到了执行。而在此时，事件并没有在队列中删除（这点在 Qt 的 Assistant 中也说得很清楚），在函数的末尾才删除了处理过了的事件，所以 hasPendingEvents()返回“false”也是非常正常的，通过对代码的调试跟踪也验证了这一点。

附录 E Qt 资源

在使用 Qt 时，如果有问题需要进行交流讨论，可以访问一些 Qt 相关的技术论坛。国外较好的 Qt 论坛有：

<http://www.qtforum.org>
<http://www.qtcentre.org>

Trolltech 公司的官方邮件列表是 qt-interest@trolltech.com。可以发送主题为“subscribe”的邮件到该邮箱订阅邮件列表，退订则发送“unsubscribe”主题的邮件到该邮箱。邮件列表的内容可以通过 nntp 服务器或 Web 方式查看，nntp 服务器地址是 [nntp://nntp.trolltech.com](http://nntp.trolltech.com)。如果需要在 nntp 新闻组中发言，可以到如下网址注册：

<http://trolltech.com/newsapply>

国内几个关于 Qt 的论坛如下：

<http://www.qtcn.org>
<http://www.qtopia.org.cn>
<http://www.qobject.com>

专门关注 Qt 的博客有：

<http://www.blogistan.co.uk/qt/>

Qt 季刊（Qt Quarterly，<http://doc.trolltech.com/qc/>）是 Trolltech 公司关于 Qt 的技术刊物，文章有一定深度。

Qt 实验室（<http://labs.trolltech.com/blogs/>）是 Trolltech 公司和客户及开源社区的交互窗口，包括正在试验的软件，开发者的博客和论坛。

精通 Qt4 编程

蔡志明 卢传富 李立夏 等编著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

软件开发与图形界面的最佳实践

精通 Qt4 编程

蔡志明 卢传富 李立夏 等编著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

精通
Qt4
编程历年Qt编程实践的精彩再现
立足理论，完全仿真实际开发

本书详细介绍了Qt的基础知识和GUI编程应用，案例翔实，内容全面，基本涵盖了Qt编程的各个方面。全书共分3篇21章，同时，在相关章节也穿插了一些重要的知识点，包括元对象系统、线程系统、对象树机制、信号/槽机制等。

初级、中级、高级三大层次完全实战

初级篇：

理论结合实践，作者将开发的前台程序都封装成精心设计的Qt4编程案例和库类函数，案例轻松而基础，适合初学者学习。

中级篇：

概念清晰，实践应用，完全仿真实际开发环境下的开发，案例循序渐进，基础部分衔接紧密。

高级篇：

内容丰富，轻松理解，完全独立开发GUI应用程序，快速提升生产力，本书附代码，可在WWW.BROADVIEW.COM.CN上下载得到。

友情提示：

阅读本书的读者需要具有基本的C++程序设计知识，毕竟Qt是用C++编写的应用程序框架。如果要学习QtScript，还需要了解JavaScript。

上架建议：QT4

网上订购：www.broadview.com.cn
定价：49.80元

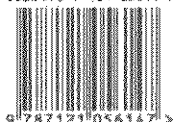


责任编辑：孙学瑛
责任美编：谢丹丹



本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

1594978-7-121-05614-7



9 787121 056147 >

定价：69.80元

前 言

两年前,当我们准备在 Linux 系统下开发 GUI 应用软件时,首先想到的就是选择一个 GUI 应用框架来简化开发。在三大 GUI 框架 GTK+、Qt 和 wxWidgets 之间,我们选择了 Qt 4 工具包。作为重量级桌面系统 KDE 多年的坚实基础,Qt 应该是经受了足够的考验。当我们准备编写自己的应用软件时,却发现图书市场上没有一本关于 Qt 4 的书籍,仅有的只是一些关于 Qt 3 的资料。由于 Qt 3 到 Qt 4 的变化很大,甚至源代码都不兼容,所以这些资料的参考价值并不是太大。于是,我们通过阅读 Qt 的 assistant 和 examples 来学习并使用 Qt 4。在逐渐掌握 Qt 4 的过程中,我们萌发了编写一本关于 Qt 4 的书来帮助初学者入门的想法。最终,在电子工业出版社博文视点资讯有限公司的大力支持下,我们的想法终于得以付诸实施。

关于 Qt

Qt 是挪威的 Trolltech 公司的旗舰产品,作为跨平台的应用程序框架,是开源的桌面系统 KDE 的基石。Google Earth, Skype, Opera, Adobe Photoshop Elements, Peforce Visual Client 等软件都是基于 Qt 写成。自 Trolltech 公司 1996 年推出 Qt 1.0 版以来,Qt 已经从 2.x, 3.x 发展到了现在的 Qt 4.3, 本书就是基于最新的 Qt 4.3 写成。因为 Qt 4 框架设计得非常优秀,在 2006 年的第 16 届 Jolt 大奖上,Qt 4 获得了类库、框架和组件类别的 Jolt 生产力奖。

和 Java 的“一次编译,到处运行”跨平台不同的是,Qt 是源代码级的跨平台,一次编写,随处编译。一次开发的 Qt 应用程序可以移植到不同的平台上,只需重新编译即可运行。Qt 支持的平台有:

- Microsoft Windows, 包括 Windows 98/NT 4.0/2000/XP/Vista;
- UNIX/X11, 包括 Linux, Sun Solaris, HP-UX, HP Tru64 UNIX, IBM AIX, SGI IRIX 等;
- Mac OS X, 支持 Mac OS X 10.3 以上版本;
- 嵌入式 Linux, 包括支持 framebuffer 的所有 Linux 平台。

Qt 还支持嵌入式系统,Qt 的嵌入式版本称为 Qtopia Core,可以在多种处理器上运行,目标操作系统通常是嵌入式 Linux。Qtopia Core 应用程序直接使用 framebuffer,而不是笨重的 X Window 系统。Qt 相关的另一个产品——Qt Jambi,则是基于 Qt 库构建的,面向 Java 程序员的应用程序框架。另外,还有一些开源的在其他语言上的 Qt 绑定,如 C#/Mono 的绑定 Qyoto, Python 的绑定 PyQt, Ruby 的绑定 QtRuby 等。有了这些产品,编写 Qt 程序不再是 C++程序员的专利了。

Qt 的发行版本有商业版和开源版。开源版遵循 QPL(Q Public License)和 GPL(GNU General Public License)协议,商业版则提供了一些特有的模块,如 Windows 平台上的 ActiveQt 框架,Oracle, DB2 等商业数据库的驱动。本书主要介绍开源版的 Qt 4.3。

阅读本书的基础

阅读本书的读者需要具有基本的 C++程序设计知识,毕竟 Qt 是用 C++编写的应用程序框架。如果要学习 QtScript,还需要了解 JavaScript。

本书的结构

本书共 21 章，每章讨论一个专题。章节安排上基本采用循序渐进、由浅到深的原则。但最后的高级篇中的章节没有很强的关联，可以按照随意的顺序阅读。每章内容及作者分述如下：

篇章	章 名	作者	内 容 简 介	页码
初级篇	第 1 章 Qt 初步实践	卢传富	建立了第一个较简单的 Qt 应用程序，在 GUI 用户界面中显示一行中文。	2
	第 2 章 对话框——QDialog	卢传富	介绍了 Qt 的对话框类 QDialog，实现了一个自定义的登录对话框，举例说明了 Qt 提供的内建对话框类的应用。	14
	第 3 章 基础窗口部件——QWidget	卢传富 蔡志明	首次引入 Qt 设计器的使用，绘制并实现了一个查找文件功能的部件，介绍了 Qt 应用程序中使用 ui 文件的基本方法以及 Qt 样式表；较深入地分析了 Qt 对象模型的一些基本知识，涉及信号和槽机制，Qt 元对象系统、属性系统和对象树机制，以及部件类型和部件的几何布局等内容。	35
	第 4 章 程序主窗口——QMainWindow	卢传富	Qt 应用程序的主窗口是由多个部件/组件构成的框架，本章通过一个简单文本编辑器的例子，介绍了主窗口的菜单、工具条、中心部件、锚接部件和状态条，并通过 Qt 设计器绘制和手写代码两种方法实现了简单文本编辑器主窗口界面的排布和管理。	87
	第 5 章 布局管理	卢传富	布局管理是 GUI 应用程序编程的一个重要方面，Qt 提供了多种布局管理部件，包括 Qt 布局管理器、分裂器、栈部件、工作空间部件和多文档区部件等。本章——介绍了这些部件，并举例说明了它们在图形用户界面编程中的应用。	121
中级篇	第 6 章 2D 绘图	蔡志明	本章内容较多，包括 Qt 的绘图要素、图形变换与坐标系统、绘图设备、图像处理、图像打印等。最后讲解了 Qt 4 图形系统的模型视图框架——Graphics View 框架。	152
	第 7 章 拖放操作与剪贴板	蔡志明	本章简要地说明了基于 MIME 的拖放操作和剪贴板的使用，关于 Graphics View 框架的拖放操作也在本章。	212
	第 8 章 文件处理	蔡志明	介绍了 Qt 的文件处理，包括基于流的文本文件和二进制文件处理，文件信息和目录操作，目录以及文件的变化监控，文件引擎的编写。	219
	第 9 章 网络	李立夏	介绍了 Qt 的网络处理，包括编写常见的 FTP、HTTP、UDP 和 TCP 程序，以及访问底层网络接口信息和扩展 Qt 网络模块功能的方法。	227
	第 10 章 多线程	李立夏	介绍了 Qt 的多线程处理，包括两方面内容：传统的线程操作，以及与 Qt 事件机制相关的操作。这一章还涉及较多的基本概念，并逐一做了介绍。	261
	第 11 章 事件机制	李立夏	介绍了 Qt 的事件处理模型，详细介绍了在 Qt 程序设计中处理事件的五种方法，并讨论了如何利用 Qt 事件机制加快用户界面响应速度。	283
	第 12 章 数据库	李立夏	介绍了 Qt 的数据库处理，重点介绍了如何在 Qt 中使用 SQL 语句进行数据库操作和如何利用 QSqlTableModel 这类高层次类进行常见的数据库编程。	297
	第 13 章 Qt 的模板库和工具类	卢传富 蔡志明	Qt 提供了丰富的模板库和工具类，本章只是介绍了部分内容。在这一章，重点介绍了 Qt 的容器类、QString 和 QVariant 类，简介了 Qt 的算法和 Qt 正则表达式的使用。	326

续表

篇章	章 名	作者	内 容 简 介	页码
高级篇	第 14 章 XML	蔡志明	对 Qt 中的三种 XML 解析方式 (DOM、SAX 和基于流的解析) 进行了比较和举例。还讲解了如何使用 API 写 XML 文件。	348
	第 15 章 模型/视图结构	蔡志明	阐述了 Qt 的模型/视图结构, 分别对模型视图的三个组成部分 (模型、视图和代理) 进行了介绍, 演示了如何自定义这些组成部分, 并简要说明了拖放以及选中操作。	366
	第 16 章 高级绘图	蔡志明	叙述了在 Qt 中如何使用 OpenGL 绘图, 对基本的 OpenGL 绘图进行了讲解, 介绍了矢量图型文件 SVG 的读写操作。	406
	第 17 章 进程间通信	李立夏	介绍进程和进程间通信的知识, 重点介绍了 Qt 中桌面环境下基于 D-Bus 的多进程应用程序开发。	421
	第 18 章 Qt 插件	蔡志明	说明了 Qt 的插件系统, 并对 Qt Designer 插件、数据库插件、风格插件进行了较详细的介绍。	442
	第 19 章 脚本——QtScript	蔡志明	这是 Qt 4.3 中引入的最新内容, 使得 Qt 能够支持 ECMAScript 脚本。本章简要地举例说明了在 Qt 中如何使用脚本, 如何将 C++ 对象暴露给脚本。	459
	第 20 章 国际化	骆艳	本章包括编码的处理, Qt Linguist 的使用步骤, 动态语言切换的内容。	468
	第 21 章 Qt 单元测试框架	蔡志明	本章阐述了如何使用 QTestLib 框架进行数据测试、GUI 测试。	478
	附录 A-E	蔡志明	附录中包括 Qt 在 Linux、Windows、Solaris 上的安装, KDevelop、Eclipse 集成开发环境的使用, qmake 的基本应用, Qt 源代码分析举例, Qt 资源。	485

如何获取源代码

由于 Qt 是跨平台的, 因此书中的内容应用能够在 Windows、Linux、UNIX 和 Mac OS 上运行, 书中的程序可能是在下列三种平台之一上编写: Windows XP/Vista、Linux (SuSE、Fedora Core 或红旗) 以及 Solaris 10 SPARC/X86。因此书中的屏幕截图可能来源于其中的任何一种操作系统。

要获取本书的源代码, 可以访问博文视点资讯有限公司网站获取:

<http://www.broadview.com.cn>。

致谢

本书在写作出版的过程中, 得到了电子工业出版社孙学瑛编辑的大力帮助, 没有她细致的工作和有益的建议, 本书难以最终出版, 在此, 作者向孙学瑛编辑表示诚挚的谢意。

问题反馈

欢迎广大读者和专家对本书提出建议和批评。如果您认为书有错误或对我们有什么建议, 可以联系 jsj@phei.com.cn。

蔡志明 卢传富 李立夏
2007 年 11 月 30 日于武汉

内 容 简 介

本书详细介绍了 Qt 的基础知识和 GUI 编程应用, 举例翔实, 内容全面, 基本涵盖了 Qt 编程的各个方面。全书共分 3 篇 21 章, 包括 Qt GUI 编程的基础知识(对话框、基础窗口部件、程序主窗口、布局管理), 中级编程(2D 绘图、拖放操作与剪贴板、文件处理、网络编程、多线程、事件机制、数据库以及 Qt 的模板库和工具类)和高级应用(XML 应用、模型/视图结构、高级绘图、进程间通信、Qt 插件和脚本应用)。同时, 在相关章节也穿插了一些重要的知识点, 包括元对象系统、属性系统、对象树机制、信号/槽机制等。

本书体系完整, 内容实用, 可以作为 Qt 初学者的入门进阶书籍, 适合具有一定开发经验的 Qt 程序员作为参考书, 也可以作为大中专院校相关专业及培训机构的教材。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有, 侵权必究。

图书在版编目(CIP)数据

精通 Qt4 编程 / 蔡志明等编著. —北京: 电子工业出版社, 2008.1
ISBN 978-7-121-05614-7

I. 精… II. 蔡… III. 软件工具—程序设计 IV. TP311.56

中国版本图书馆 CIP 数据核字(2007)第 193845 号

责任编辑: 孙学瑛

印 刷: 北京天宇星印刷厂

装 订: 北京牛山世兴印刷厂

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 850×1168 1/16 印张: 32.75 字数: 819 千字

印 次: 2008 年 1 月第 1 次印刷

印 数: 5000 册 定价: 69.80 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zlt@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。